University of Alberta

Bridging Fault Diagnosis in CMOS Circuits

by

Michael Olson

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta
Spring 1997

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced with the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-21198-3

Canada

University of Alberta

Library Release Form

Name of Author: Michael Olson

Title of Thesis: Bridging Fault Diagnosis in CMOS Circuits

Degree: Master of Science

Year this Degree Granted: 1997

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Mike Olson

Michael Olson
4211 50 Avenue
Ponoka, AB
Canada, T4J 1C2

Date: Oct. 21, 1996.

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Bridging Fault Diagnosis in CMOS Circuits** submitted by Michael Olson in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. Xiaoling Sun

Dr. Bruce Cockburn

Dr. Nelson Durdle

Dr. M. Tamer Ozsu

Date: Oct. 21, 94.

Abstract

This thesis presents two new algorithms for diagnosing gate-level bridging faults in CMOS circuits. One algorithm uses the single fault assumption, while the other algorithm is applicable to circuits containing multiple faults. The new single fault algorithm provides a significantly higher diagnostic resolution than that provided by a well-known algorithm described by Chakravarty.

A set of design recommendations is given which improves the I_{DDQ} testability of the circuit under test. Improving the testability of a circuit enables a higher diagnostic resolution to be obtained. Recommendations are also given to maximize the performance of the algorithm by minimizing the computer memory and CPU time required to perform a diagnosis run.

Extensive computer simulations are performed to illustrate the merits and feasibility of the new algorithms using Berkeley and ISCAS85 benchmark circuits.

Acknowledgements

I would like to thank my supervisor, Dr. Xiaoling Sun, for her guidance and support throughout this research project. The financial assistance received from her research grant is also gratefully acknowledged.

I would also like to thank Wes Tutak, Alvin Poon, and Ronald Fung for their assistance with the programming aspects of this research. Thanks also go out to Norman Jantz for providing technical assistance, and Dave White for proofreading my thesis.

Finally, I would like to thank my parents for providing support and understanding throughout my university education.

Contents

1 Introduction 1

2 Background and Literature Review 7

2.1 CMOS Technology . . . 7

2.1.1 MOS Transistors . . . 7

2.1.2 CMOS Logic Structures . . . 9

2.2 Defects and Fault Modelling . . . 10

2.2.1 The Stuck-at Fault Model . . . 10

2.2.2 Bridging Defects and Bridging Faults . . . 11

2.3 Voltage-Based Testing . . . 13

2.3.1 Detection of Stuck-at Faults . . . 13

2.3.2 Bridging Fault Models and Fault Detection . . . 14

2.4 I_{DDQ} Testing . . . 16

2.4.1 Bridging Fault Detection . . . 17

2.4.2 Limitations of I_{DDQ} Testing . . . 20

2.5 Fault Diagnosis in Digital Circuits . . . 21

3 New Algorithms 29

3.1 Testing and Diagnosis Process . . . 29

3.2 Fault Assumptions . . . 31

3.3 Definitions and Declarations . . . 32

3.4 Single Fault Diagnosis . . . 35

3.4.1 Single Fault Algorithm . . . 36

3.4.2 Single Fault Diagnosis Example . . . 37

3.5 Multiple Fault Diagnosis . . . 41

3.5.1 Multiple Fault Algorithm . . . 42

3.5.2 Multiple Fault Diagnosis Example . . . 45

3.6 Complexity Analysis of Algorithms . . . 48

3.6.1 Single Fault Algorithm . . . 48

3.6.2 Multiple Fault Algorithm . . . 49

3.7 Factors Affecting Algorithm Performance . . . 50

3.7.1 Design Recommendations . . . 51

3.7.2 Efficiency Improvement Using Test Vector Re-ordering . . . 52

3.7.3 Effects of Feedback Fault Assumptions . . . 53

3.7.4 Effects of Using Different Test Sets . . . 54

3.8 Diagnosis Beyond Algorithms 56
3.8.1 Single Fault Algorithm Output 56
3.8.2 Multiple Fault Algorithm Output 57

4 Simulations **59**
4.1 Simulation Environment 59
4.2 Simulation Goals and Assumptions 61
4.3 Simulation Results 62
4.3.1 Sample I_{DDQ} Test Result Generation 62
4.3.2 Simulations Using the Single Fault Algorithm 65
4.3.3 Simulations Using the Multiple Fault Algorithm 71
4.4 Implementation 75
4.4.1 BRIDGE Classes 76
4.4.2 Diagnosis Implementation 81

5 Conclusion **84**

Bibliography **87**

Appendices **92**

A Algorithm Flowcharts **92**
A.1 Single Fault Algorithm Flowchart 93
A.2 Multiple Fault Algorithm Flowcharts 95

B BRIDGE User's Manual **100**

C Single Fault Algorithm Simulation Tables **103**
C.1 Single Fault Diagnosis Using Single Fault Patterns 104
C.2 Comparative Diagnosis Simulations 120
C.3 Single Fault Diagnosis Using Double Fault Patterns 129

D Multiple Fault Algorithm Simulation Tables **137**
D.1 Multiple Fault Diagnosis Using Single Fault Patterns 138
D.2 Multiple Fault Diagnosis Using Double Fault Patterns 154

List of Figures

1.1 Logic Testing Architecture 2
1.2 I_{DDQ} Testing Architecture 4

2.1 MOS Transistors 8
2.2 Static CMOS Architecture 9
2.3 Transistor-Level Bridging Fault 12
2.4 Gate-Level Bridging Fault 12
2.5 Two Classes of Bridging Faults 13
2.6 Detection of a Stuck-at 1 Fault on Line F 14
2.7 Bridging Fault in CMOS Inverters 18
2.8 Activating a Bridging Fault 19
2.9 Feedback Bridging Fault 20

3.1 Circuit Current During I_{DDQ} Test Session 33
3.2 CUT for Example Diagnosis Run 38
3.3 Feedback Faults on Inverters 51

4.1 Overview of Simulation Environment 60
4.2 Faults Eliminated After 90 Test Vectors for c432 64
4.3 Faults Eliminated After 90 Test Vectors for c5315 64
4.4 Flow Chart of BRIDGE 77
4.5 Gate Structure 78
4.6 Circuit Building Block 79
4.7 Event List Structure 80
4.8 Structure of DynamicArray 81

A.1 Flowchart for Single Fault Algorithm 94
A.2 Flowchart for Procedure CLASSIFY 96
A.3 Flowchart for Procedure UNDETECTED 97
A.4 Flowchart for Procedure AP_UNIQUE 98
A.5 Flowchart for Procedure NTUPLE 99

List of Tables

3.1 Nodes and Faults for Example Circuit 38
3.2 Circuit Information for Diagnosis Example 38
3.3 Logic Simulation and I_{DDQ} Test Results for Diagnosis Example . . . 39
3.4 Input to NTUPLE Procedure for Multiple Fault Example 46
3.5 Results of NTUPLE Procedure for Multiple Fault Example 48

4.1 Characteristics of Benchmark Circuits 61
4.2 Number of Test Vectors used for Berkeley Benchmark Simulations . . 65
4.3 LFSR Polynomials Used for ISCAS85 Benchmark Simulations 66
4.4 Single Fault Algorithms Using Single Fault Patterns 67
4.5 Results for Comparative Simulations 68
4.6 Single Fault Algorithm Using Double Fault Patterns 70
4.7 Multiple Fault Algorithm Using Single Fault Patterns 71
4.8 Multiple Fault Algorithm Using Double Fault Patterns 73

C.1 Results for Benchmark f2 104
C.2 Results for Benchmark rd53 105
C.3 Results for Benchmark rd73 106
C.4 Results for Benchmark sao2 107
C.5 Results for Benchmark bw 108
C.6 Results for Benchmark rd84 109
C.7 Results for Benchmark 9sym 110
C.8 Results for Benchmark c432 111
C.9 Results for Benchmark c499 112
C.10 Results for Benchmark c880 113
C.11 Results for Benchmark c1355 114
C.12 Results for Benchmark c1908 115
C.13 Results for Benchmark c2670 116
C.14 Results for Benchmark c3540 117
C.15 Results for Benchmark c5315 118
C.16 Results for Benchmark c7552 119
C.17 Results for Benchmark c432 120
C.18 Results for Benchmark c499 121
C.19 Results for Benchmark c880 122
C.20 Results for Benchmark c1355 123
C.21 Results for Benchmark c1908 124

C.22 Results for Benchmark c2670 125
C.23 Results for Benchmark c3540 126
C.24 Results for Benchmark c5315 127
C.25 Results for Benchmark c7552 128
C.26 Results for Benchmark f2 129
C.27 Results for Benchmark rd53 129
C.28 Results for Benchmark rd73 130
C.29 Results for Benchmark sao2 130
C.30 Results for Benchmark bw 131
C.31 Results for Benchmark rd84 131
C.32 Results for Benchmark 9sym 132
C.33 Results for Benchmark c432 132
C.34 Results for Benchmark c499 133
C.35 Results for Benchmark c880 133
C.36 Results for Benchmark c1355 134
C.37 Results for Benchmark c1908 134
C.38 Results for Benchmark c2670 135
C.39 Results for Benchmark c3540 135
C.40 Results for Benchmark c5315 136
C.41 Results for Benchmark c7552 136

D.1 Results for Benchmark f2 138
D.2 Results for Benchmark rd53 139
D.3 Results for Benchmark rd73 140
D.4 Results for Benchmark sao2 141
D.5 Results for Benchmark bw 142
D.6 Results for Benchmark rd84 143
D.7 Results for Benchmark 9sym 144
D.8 Results for Benchmark c432 145
D.9 Results for Benchmark c499 146
D.10 Results for Benchmark c880 147
D.11 Results for Benchmark c1355 148
D.12 Results for Benchmark c1908 149
D.13 Results for Benchmark c2670 150
D.14 Results for Benchmark c3540 151
D.15 Results for Benchmark c5315 152
D.16 Results for Benchmark c7552 153
D.17 Results for Benchmark f2 154
D.18 Results for Benchmark rd53 154
D.19 Results for Benchmark rd73 155
D.20 Results for Benchmark sao2 155
D.21 Results for Benchmark bw 156
D.22 Results for Benchmark rd84 156
D.23 Results for Benchmark 9sym 157
D.24 Results for Benchmark c432 157

D.25 Results for Benchmark c499 158
D.26 Results for Benchmark c880 158
D.27 Results for Benchmark c1355 159
D.28 Results for Benchmark c1908 159
D.29 Results for Benchmark c2670 160
D.30 Results for Benchmark c3540 160
D.31 Results for Benchmark c5315 161
D.32 Results for Benchmark c7552 161

Chapter 1

Introduction

The trend in integrated circuit design is to smaller and more densely integrated systems. Currently, digital circuits are being fabricated using sub-micron technology. The smaller component dimensions and increased circuit density have made the task of detecting and analysing failures in digital circuits a difficult problem. The demand for higher chip yields and more reliable circuit operation has heightened the importance of detecting failures during the design process of Very Large Scale Integrated (VLSI) circuits.

The analysis of faulty digital circuits can be broken down into two basic categories: testing and diagnosis [3, page 1]. A testing process provides a "pass/fail" signal indicating whether the circuit is functioning correctly or incorrectly. If a circuit fails a test, a diagnosis process may be required. In a diagnosis process, the location of defective components or areas in the circuit are identified. This thesis introduces a new diagnosis process.

Diagnosis may be performed at the system level, the board level, the gate level, or at a lower level such as the transistor level. For digital systems already in field operation, board-level diagnosis is useful for determining a defective system component, and replacing it quickly. When a defect is located, the entire circuit board or chip is replaced.

In design and manufacturing processes, a higher diagnostic resolution, such as gate-level or transistor-level diagnosis, is required. For these types of diagnosis, the goal is to find out exactly which logic element is causing the circuit to behave in a faulty manner. If the location of the defect can be localized to a specific logic gate or

signal line, possible manufacturing errors or design mistakes can be identified. This thesis focuses on diagnosis performed at the gate level.

Before diagnosis is performed, testing is done to determine whether the circuit is operating correctly. Testing methods can be grouped into two main categories: voltage testing methods and parametric testing methods. In *voltage testing*, the tests determine whether the correct output responses are produced when particular inputs are applied to the circuit. In *parametric testing*, a parameter of the circuit, such as the quiescent current or the maximum propagation delay, is measured. A faulty circuit is identified if it does not operate within its expected parameters. A circuit that passes a voltage test may still fail a parametric test [18].

Most voltage testing, or logic testing, methodologies are designed to analyse the circuit's response to large amounts of input stimuli in a short amount of time. A typical voltage testing architecture consists of a mechanism to provide test patterns (inputs) to the *circuit under test* (CUT), and a method to collect and evaluate the output responses. Many techniques have been proposed for both test pattern generation and output response analysis [3]. Figure 1.1 illustrates a typical logic testing architecture.

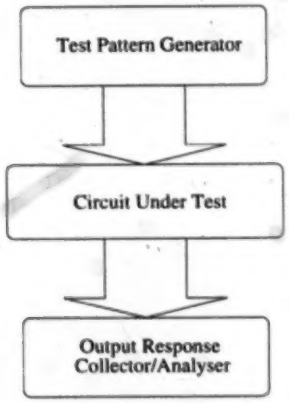


Figure 1.1: Logic Testing Architecture

Erroneous circuit responses are usually attributed to defects in the circuit. Defects can be formed due to imperfections in a manufacturing process, from mishandling during shipping, from environmental exposure, or from simple wear out during regular circuit operation. Most testing methods employ *fault models* [3, page 93] to avoid

directly handling the large number of possible physical defects in a CUT. A fault model is a set of rules and assumptions which describe the effects that defects have on digital circuits. When fault models are used, the testing problem becomes one of testing for the presence of a manageable number of logical faults rather than a large number of physical defects.

The most well-known fault model is the *stuck-at fault model* [3, page 110]. In the stuck-at fault model, the logic gates in the CUT are assumed to be fault-free. Any defects in the circuit are assumed to cause one or more of the signal nets between the logic gates to be permanently "stuck" at either a logic 1 value, or a logic 0 value. Due to its simplicity and its success in modelling many types of physical defects, the stuck-at fault model has been in wide use in testing applications for over three decades.

Once a fault model is specified for a testing architecture, a method of test pattern generation must be chosen to provide the input stimulus to the CUT. Ideally, in order to detect all faults in an n-input CUT, all possible input patterns (an exhaustive test set) should be used. For circuits with a large number of inputs (i.e. > 23), exhaustive tests are impractical due to the large amount of time required to apply the tests and the large amount of memory required to store the test vectors. For example, it is common for commercial circuits to have 100 or more inputs. For a 100 input circuit, if two million test patterns are applied to the circuit every second, it will take 7.3×10^{18} years to complete an exhaustive test session. In practice, tests should only take seconds as this helps keep testing costs down. The subject of test pattern generation deals with finding a high-quality, non-exhaustive test set which uses the fewest number of test vectors to detect the maximal number of modelled faults.

The quality or *grade* of a test set is rated according to the *fault coverage* that it provides [3, page 131]. The fault coverage of a test set is the ratio of the number of faults which are detected by the set, to the total number of possible faults in the circuit. Fault coverage is computed by a process called *fault simulation* [3, page 131]. Fault simulation consists of simulating a CUT in the presence of all the faults specified by a fault model and recording or analysing the results. The results of fault simulation are then compared to those obtained from a fault-free simulation of the CUT to determine the fault coverage. It should be pointed out that if a test set

provides 100% fault coverage, it may still fail to detect defects which are not modelled by the fault model [3, page 131]. Subsequently, testing engineers often employ some form of parametric testing along with logic testing in order to increase the overall testing quality.

A commonly used brand of parametric testing for Complementary Metal Oxide Semiconductor (CMOS) circuits is I_{DDQ} testing [37]. I_{DDQ} testing (quiescent current testing) involves monitoring the steady-state current that flows in a CMOS circuit. Since fault-free CMOS circuits allow very little current flow in the steady-state [37], a high level of current indicates that the circuit is faulty. The new diagnosis methods presented in this thesis use the results from I_{DDQ} tests which are performed on the CUT. Figure 1.2 illustrates an I_{DDQ} testing architecture.

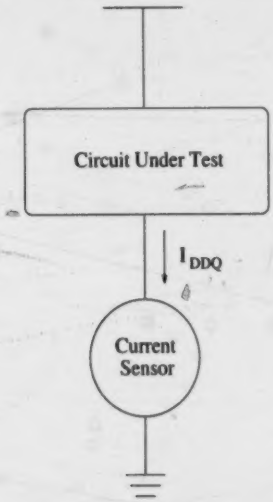


Figure 1.2: I_{DDQ} Testing Architecture

It has been shown that common defects which occur in CMOS circuits are unintentional shorts or bridges between signal lines [11, 19, 27], and that many bridging defects do not cause predictable logic errors to occur in the circuit [19]. Consequently, bridging fault models are used to describe shorts in a CUT, and can be used at mask, transistor, gate, and higher levels of circuit representations. This thesis deals with the problem of diagnosing gate-level bridging faults in CMOS circuits.

Since the number of two-node bridges that can exist in a circuit is far greater than the number of possible stuck-at faults, and since a single test vector may detect a

large number of bridging faults, the diagnosis of bridging faults is the more difficult problem. Previous diagnosis methods have used *fault dictionaries* [3, page 7] to relate patterns of faulty circuit outputs to specific bridging faults [31]. The fact that some bridging faults do not cause logic errors, as well as the fact that a large amount of memory is required to store all of the possible faults in the CUT, makes this type of diagnosis infeasible.

Another method of bridging fault diagnosis is the use of algorithms to process either voltage-based test results [13] or I_{DDQ} test results [14, 15]. A fast algorithm which employs I_{DDQ} test results to identify the set of possible two-node bridging faults is given in [14]. Although the algorithm is very efficient, it relies on the assumption that only one bridging fault is present in the circuit during the test session (single fault assumption). The algorithm is also overly pessimistic in the way that it handles bridging faults which create feedback paths in the CUT, thus it produces a poor diagnostic resolution.

This thesis presents two new diagnosis algorithms to diagnose gate-level bridging faults in CMOS circuits. The algorithms use the results from an I_{DDQ} test session to determine the fault(s) that can be responsible for causing the I_{DDQ} failures. The algorithms offer an advantage over the algorithm given in [14], in that more faults are eliminated during a diagnosis run. This allows for better diagnostic resolution than can be obtained in [14]. Furthermore, one of the new algorithms is not limited by the single fault assumption, and CUTs containing more than one bridging fault can be diagnosed correctly. The new algorithms are easy to implement, and no circuit-level fault simulation is required. Moreover, no previous layout information about the CUT is required, which makes it possible to diagnose a circuit using only the connectivity information. This thesis is organized as follows.

Chapter 2 provides the necessary background information on digital circuit testing and diagnosis. General testing and diagnosis techniques are discussed, and some basic terminology is defined. A review and summary of some previously presented diagnosis methods is also given.

Chapter 3 introduces the new diagnosis algorithms. The general theory behind the algorithms is given. Definitions and terminology used in the algorithms are presented. Detailed examples illustrating the operation of each new algorithm are given.

Subsequent diagnostic methods are also discussed in order to show how the new algorithms can be used as part of a complete diagnostic system. Different factors and assumptions which affect the resolution and performance provided by the algorithms are also discussed.

Chapter 4 describes the simulations performed to test and justify the diagnosis algorithms. The simulation environment is described. Results of the simulations are presented, and conclusions about the effectiveness of the algorithms are drawn. A description of the simulator and the way that it implements the diagnosis algorithms is given.

Finally, Chapter 5 draws conclusions about the performance achieved by the algorithms and summarizes the entire thesis.

Chapter 2

Background and Literature Review

This chapter presents an overview of some of the topics relating to the diagnosis of bridging faults in CMOS circuits. CMOS digital circuit architecture is described. The concept of fault modelling is discussed. An overview of bridging faults is given. Basic digital circuit testing methods, namely voltage testing and I_{DDQ} testing, are reviewed. Previously presented diagnosis methods for digital circuits are also discussed and evaluated.

2.1 CMOS Technology

CMOS technology is currently the dominant technology used to construct digital circuits. The following paragraphs describe the basic theory behind CMOS technology.

2.1.1 MOS Transistors

CMOS circuits are built using two different kinds of transistors: PMOS transistors and NMOS transistors (see figure 2.1). For both types of transistors, the voltage applied to the transistor gate controls the amount of current that flows from the drain terminal to the source terminal. Equations 2.1 to 2.4 describe the relationship between the gate voltage and the resulting current flow for both PMOS and NMOS transistors [30, page 145].

For PMOS transistors,

$$I_{SD} = k_p \frac{W}{L}(V_{SG} - \mid V_T \mid)^2 \quad if \quad V_{SD} \geq V_{SG} - \mid V_T \mid \tag{2.1}$$

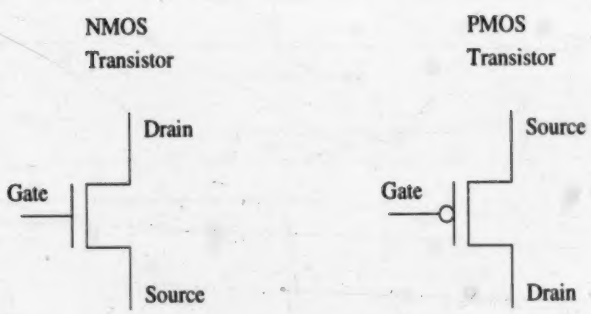


Figure 2.1: MOS Transistors

$$I_{SD} = k_p \frac{W}{L}(2(V_{SG} - \mid V_T \mid)V_{SD} - V_{SD}^2) \quad if \quad V_{SD} < V_{SG} - \mid V_T \mid \tag{2.2}$$

where,

$k_p = \frac{\mu_p C_o}{2}$,

μ_p is the hole mobility,

C_o is the gate capacitance per unit area,

V_{SG} is the source-to-gate voltage,

V_{SD} is the source-to-drain voltage,

V_T is the threshold voltage,

W is the width of the transistor channel,

L is the length of the transistor channel,

I_{SD} is the current flow from source-to-drain.

For NMOS transistors,

$$I_{DS} = k_n \frac{W}{L}(V_{GS} - V_T)^2 \quad if \quad V_{DS} \geq V_{GS} - V_T \tag{2.3}$$

$$I_{DS} = k_n \frac{W}{L}(2(V_{GS} - V_T)V_{GS} - V_{GS}^2) \quad if \quad V_{DS} < V_{GS} - V_T \tag{2.4}$$

where,

$k_n = \frac{\mu_n C_o}{2}$,

μ_n is the electron mobility,

C_o is the gate capacitance per unit area,

V_{GS} is the gate-to-source voltage,

V_{DS} is the drain-to-source voltage,

V_T is the threshold voltage,

W is the width of the transistor channel,

L is the length of the transistor channel,

I_{DS} is the current flow from drain-to-source.

In digital CMOS circuits, when a high logic level is applied to the gate of an NMOS transistor, a conducting channel will be formed from the drain to the source and the transistor will allow a large amount of current to flow. The same holds true for a PMOS transistor if a low logic level is applied to the gate.

2.1.2 CMOS Logic Structures

In a fully complementary MOS (FCMOS) logic block, a network of PMOS transistors is connected to the power rail and a complementary network of NMOS transistors is connected to ground (see figure 2.2). During the steady state, a low resistance path is created from one of the power rails (Vdd or GND) through one of the two transistor networks (PMOS or NMOS) to the output of the logic block. The path that is set up depends upon the logic values assigned to the input line(s) of the logic block. Logic values are set on the output node by either charging or discharging the node capacitance.

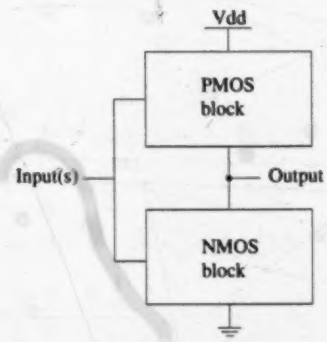


Figure 2.2: Static CMOS Architecture

An important characteristic of CMOS logic blocks is that they allow almost no current to flow during the steady state. This is because one of the two transistor networks will "open" the current path between Vdd and GND. Although large amounts

of current can flow in the transient state immediately after changing the circuit inputs, the only current that flows during the steady state is the transistor leakage current. This small current flow means that CMOS circuits have very low power dissipation [37].

2.2 Defects and Fault Modelling

A *fault model* is a set of rules or assumptions which describe the logical behaviour or reaction that a digital circuit has to physical defects [1]. The advantages of using logical faults to represent defects are: the complexity of the testing problem is reduced since many different physical defects are often represented by the same logical fault; some logical faults are technology-independent as their fault model can be applied to many different technologies; tests derived for logical faults can often be used to detect defects whose effects on circuit behaviour are not completely understood [3, page93].

2.2.1 The Stuck-at Fault Model

A *structural* fault model is one in which the effect of each fault is to modify the interconnections between circuit components [3, page93]. The most common structural fault model is the stuck-at fault model [3, page 110]. In this fault model, all logic gates are assumed to be fault-free, while defects are assumed to cause the signal lines between logic gates to be permanently stuck at either a logic 1 or a logic 0, regardless of the input stimulus to the CUT. For example, a short between either Vdd or GND and a signal line will cause that signal line to behave as though it is permanently stuck at a logic 1 or 0, respectively.

The stuck-at fault model has the following desirable qualities. It represents many different physical defects. It can be used to analyse all digital circuits, regardless of what technology is used during fabrication. Test sets designed for stuck-at faults will also detect many other kinds of faults. Finally, the total number of possible stuck-at faults can be easily calculated, as each line in the circuit has two possible faulty conditions. Therefore, it is easy to determine the fault coverage or quality of any set of input test vectors. Moreover, the number of faults that have to be investigated during a test session can be reduced using fault-collapsing techniques [3, page 106].

2.2.2 Bridging Defects and Bridging Faults

In VLSI circuits, signal nets are routed through narrow channels over relatively long distances. When there is a high density of nets routed in one channel, there is a possibility that two or more nets can become unintentionally shorted together. An unintentional short or bridge may result due to a problem during manufacturing, such as spot defects existing on the manufacturing masks.

Previous studies have shown that bridging defects are the most common type of defect in CMOS circuits [11, 19, 27, 38]. In [11] it is stated that 30% to 50% of the total defects in a circuit can be classified as unintentional bridges. The frequent occurance of bridging defects makes their detection and diagnosis an active area of current research.

Bridging defects are commonly represented as schematic level bridging faults. Bridging faults can be classified depending upon their locations in a circuit and the logical relationship between the fault's component nodes. The following paragraphs review the classes which are commonly used to group and describe bridging faults.

Bridging Fault Levels

A transistor-level bridge is one in which the bridged nodes are internal to a logic gate. Figure 2.3 illustrates a transistor level bridging fault inside a CMOS NAND logic gate.

Today digital circuits are often realized using standard cell libraries which contain pre-defined standard logic gates, such as AND gates, OR gates, and inverters. Gate-level bridging faults are faults which exist between the input nodes and the output nodes of the logic gates in a circuit. Previous studies have reported that gate-level faults are the most common type of bridging fault, as the nets between gate outputs and gate inputs are more prone to spot defects than other types of nets [34]. Figure 2.4 illustrates a gate-level bridging fault.

Feedback and Non-Feedback Bridging Faults

The two main classes of bridging faults are feedback bridges and non-feedback bridges. In a feedback bridge, a signal path exists between the two component nodes of the

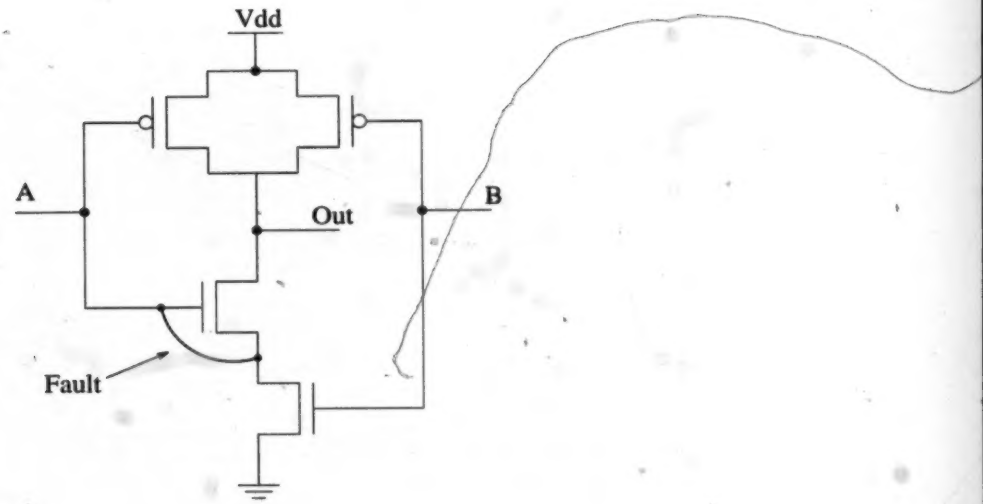


Figure 2.3: Transistor-Level Bridging Fault

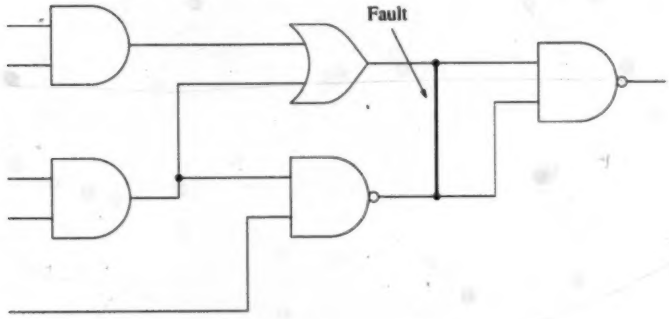


Figure 2.4: Gate-Level Bridging Fault

fault. Therefore, the logic value of one node can depend on the logic value of the other node, even in a fault-free circuit. In a non-feedback bridge, no signal path exists between the bridged nodes other than the fault itself. Figure 2.5 illustrates the difference between feedback and non-feedback faults at the gate level.

Feedback and non-feedback bridging faults have different logical effects on a circuit. Feedback faults transform combinational logic circuits into asynchronous sequential logic circuits. These sequential circuits may exhibit stable states [25]. If the signal path between the fault's component nodes contains an odd number of signal inversions, the bridging fault may cause oscillations to occur. When the rise and fall times of the node logic levels involved with the fault are negligible compared to the

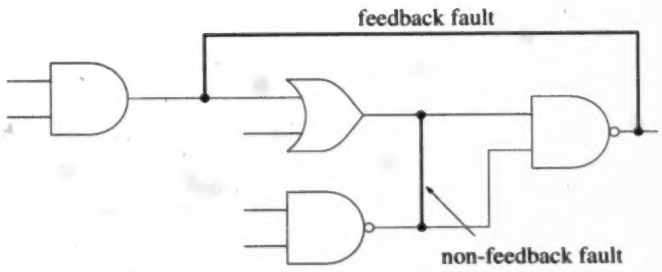


Figure 2.5: Two Classes of Bridging Faults

propagation delay on the signal path between the nodes, the oscillations will have a period equal to the propagation delay [25].

Since the nodes in a non-feedback fault do not influence each other in the fault-free case, they do not cause problems such as stable states and oscillations associated with feedback faults. Non-feedback faults will only influence the logic level of the bridged nodes themselves and all gates which are subsequently driven by these nodes. The logic function that a bridge implements on a pair of nodes depends on the technology used for circuit construction.

2.3 Voltage-Based Testing

In voltage-based testing methods, a circuit is declared faulty if it produces incorrect output data for a corresponding set of inputs. This section describes how voltage-based testing is used to detect both stuck-at faults and gate-level bridging faults.

2.3.1 Detection of Stuck-at Faults

Two actions must be performed in order to detect a stuck-at fault. The fault must be activated by an input test vector, and the effect of the fault must be propagated to a primary circuit output for observation. Only certain test vectors allow both activation and propagation to take place. Therefore, to thoroughly test a circuit for stuck-at faults, a set of input test vectors is required that will activate and propagate the effects of every stuck-at fault to a primary output at least once during the test session.

Figure 2.6 illustrates the propagation of a stuck-at fault to a circuit output. If

signal line F is stuck-at a logic 1, the effect of this fault must be propagated to output node L in order for the fault to be detected. If the input vector to the circuit is as shown, (A B C D E) = (0 1 0 1 0), line F has a fault-free value of logic 0 and line L has a fault-free value of logic 1. When line F is stuck-at 1 and the test vector is applied to the CUT, a faulty value of logic 0 is observed on line L. Consequently, the "F stuck-at 1" fault is said to be detected by this test vector. Conversely, if the input vector to the circuit is (A B C D E) = (0 1 0 1 1), the fault-free value at line L is no different from the value at line L when line F is stuck-at 1. In this case, the effects of the fault are not propagated to the output even though the stuck-at fault is activated. Therefore, this test vector does not test for the "F stuck-at 1" fault.

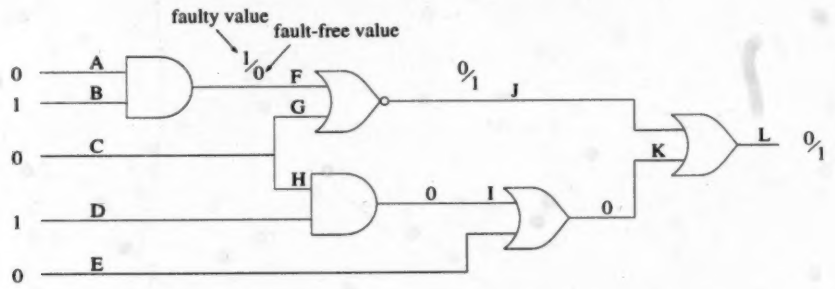


Figure 2.6: Detection of a Stuck-at 1 Fault on Line F

2.3.2 Bridging Fault Models and Fault Detection

Despite its popularity, the stuck-at fault model is not always good at describing faulty circuit behaviour. For example, the effects of bridging defects in CMOS circuits are not modelled accurately using stuck-at techniques. This subsection reviews two fault models that have been used to model the logical circuit behaviour due to bridging faults.

Wired-AND/OR Bridging Fault Model

The wired-AND/OR bridging fault model [1] assumes that bridges perform either a wired-AND or a wired-OR logic function. For a wired-AND function, if either bridged node is driven to a logic zero, both nodes will be pulled down to a logic 0. Similarly, for the wired-OR function, if either node is driven to a logic 1, both nodes are pulled

up to a logic 1. As with the stuck-at fault model, the effect of the bridge must be propagated to a circuit output in order for the bridge to be detected.

While the wired-AND/OR bridging fault model performs adequately for older technologies such as TTL and ECL, it is inadequate for describing the behaviour of bridges in CMOS circuits [20]. Bridges in CMOS circuits can create intermediate logic values on the bridged nodes instead of a clear logic 0 or logic 1. Since different logic gates have different logic threshold levels [27], these intermediate values may be interpreted differently by different gates. For example, if two bridged nodes are at 2.3 volts, a gate input with a transition threshold of 2.0 volts will treat this as a logic 1. However, a gate input with a threshold of 3.0 volts will treat 2.3 volts as a logic 0. Consequently, it is not guaranteed that the bridge will implement a simple AND/OR operation.

Feedback bridging faults can present further difficulties for the wired AND/OR fault model. Due to the problems of unwanted states and oscillations, it can be difficult to find an input test set that will propagate the effects of a feedback fault to the circuit outputs.

Voting Fault Model

The inability of the wired-AND/OR model to accurately describe CMOS bridging faults has led to the development of a more complex bridging fault model known as the *voting model* [4]. For gate-level bridging faults, the voting model works as follows.

The voting model assumes that one of the two conducting transistor networks (PMOS and NMOS) involved in a bridge will have more driving strength than the other. Consequently, the bridged nodes will never sit at an intermediate logic value. If the output nodes of two logic gates are shorted together, one of the shorted nodes will be dominant over the other node. The function of the voting model is to estimate which of the networks will have more current driving capability, and based on this estimate, predict which of the nodes will be dominant.

Different criteria can be used by the voting model to determine which transistor network will "win" a vote and set the logic value of the bridged nodes. One method is to rank all of the logic gates in the CUT according to current driving strength based upon previously performed-circuit level simulations. When a bridge occurs between

the output nodes of two gates, the gate with the higher recorded driving strength is assumed to win the vote and determine the logic value on the bridge. A second method is to employ wired-AND or wired-OR assumptions. Finally, the most complex method is to derive expressions using the number of inputs for each gate, and the number of conducting transistors activated in each gate by every test vector in the test set. Since the number of conducting transistors is proportional to the driving strength, the network having the most conducting transistors is assumed to win the vote.

A drawback to using the voting model is that it can require numerous computations and simulations. Since the driving strength of a transistor network is influenced by many factors such as transistor sizing, the type of CMOS technology, and the variation of transistor characteristics over different wafers [4], the use of circuit level simulators, such as SPICE, is necessary to determine the driving strength of different transistor networks [27].

Once it is determined which transistor network sets the logic level on a bridge, the effect of that bridge must be propagated to a circuit output in order for the bridge to be detected. The problem of finding an appropriate test vector to perform this propagation can be challenging. If the fault in question is a feedback fault, the problem of test vector selection becomes even more difficult. These problems are discussed more thoroughly in [4].

The complexity of the voting model illustrates the difficulty in creating a simple fault model which accurately describes the logical circuit behaviour due to bridging defects. Moreover, it has been shown in [19] that the difficulty in modelling bridging faults increases if the faults have non-zero resistive values. To avoid the problems inherent with logic testing, a different testing method called I_{DDQ} testing can be employed. The basics of I_{DDQ} testing are described in the following section.

2.4 I_{DDQ} Testing

I_{DDQ} testing (quiescent current testing) [37] is a parametric testing method that is applicable to CMOS circuits. I_{DDQ} testing involves monitoring the steady state current that flows between the power and ground rails of a CMOS circuit. If the

amount of steady state current flowing in a CUT is more than a prespecified threshold level, the CUT is declared faulty.

It was shown in figure 2.2 that a CMOS logic block contains a PMOS network and and NMOS network, only one of which provides a low-resistance path to the output during the steady state. As a result of this architecture, a defect-free CMOS logic block allows almost no steady state current to flow. If a CMOS circuit contains a defect, an abnormally high steady state current may flow depending upon both the nature of the defect, and the input stimulus applied to the circuit. The detection of high steady state currents can be used to indicate the presence of a defect. It has been shown in previous studies that zero-defect testing is not possible without performing I_{DDQ}-based tests [43].

2.4.1 Bridging Fault Detection

Due to the unique nature of CMOS circuits, I_{DDQ} tests can detect bridges that are not detectable using voltage-based testing methods. This is illustrated using the two CMOS inverters in figure 2.7. In figure 2.7, there is an unintentional bridge between the output lines of the two inverters. In a fault-free case, and with the specified inputs, the top inverter will pass a logic 1 to its output while the bottom inverter will pass a logic 0 to its output. Moreover, there is no low resistance path from Vdd to GND in either inverter.

After the introduction of the bridging fault, the logic level at both inverter outputs is not readily known. The output voltage will depend on the relative driving strengths of the PMOS transistor in the top inverter, and the NMOS transistor in the bottom inverter. Consequently, the fault may or may not introduce an observable faulty logic value on the shorted nodes.

The dashed line in figure 2.7 shows that a low resistance path exists between Vdd and GND due to the bridge. Even if the output node is at a voltage between Vdd and GND, the low resistance path clearly indicates the presence of the fault.

To illustrate how I_{DDQ} current changes in the presence of a bridge, consider the following analysis. The equation for the steady-state current for the circuit in figure 2.7 changes depending upon which transistor has more driving strength. For

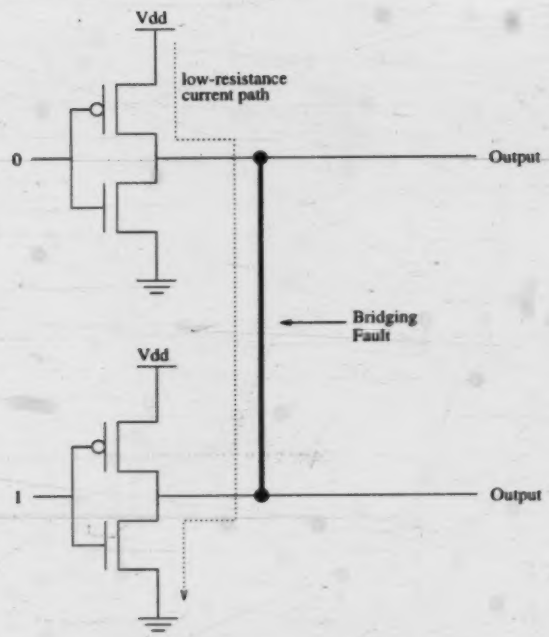


Figure 2.7: Bridging Fault in CMOS Inverters

example, if the NMOS transistor has more driving strength, the equation becomes:

$$I_D = k_p \frac{W}{L}(V_{SG_p} - V_T)^2 = k_n \frac{W}{L}(2(V_{GS_n} - V_T)V_{OutputS_n} - V_{OutputS_n}^2). \tag{2.5}$$

If the PMOS transistor has more driving strength, the equation is:

$$I_D = k_n \frac{W}{L}(V_{GS_n} - V_T)^2 = k_p \frac{W}{L}(2(V_{SG_p} - V_T)V_{S_pOutput} - V_{S_pOutput}^2). \tag{2.6}$$

If both transistors have close to equal driving strengths the equation becomes:

$$I_D = k_p \frac{W}{L}(2(V_{SG_p} - V_T)V_{S_pOutput} - V_{S_pOutput}^2) = k_n \frac{W}{L}(2(V_{GS_n} - V_T)V_{OutputS_n} - V_{OutputS_n}^2). \tag{2.7}$$

In all these situations, I_D will be much larger than it is in the fault-free circuit. Therefore, if I_{DDQ} testing is performed on this circuit, this bridging fault is easily detected.

Non-Feedback Bridging Fault Detection

The ease of detection of a CMOS bridge depends upon the nature of the fault itself. Non-feedback bridging faults can be easily tested with I_{DDQ} methods. The testing

process is a matter of finding an input vector that will activate the bridge while the steady-state current in the circuit is monitored. Activating a non-feedback bridging fault is illustrated in figure 2.8.

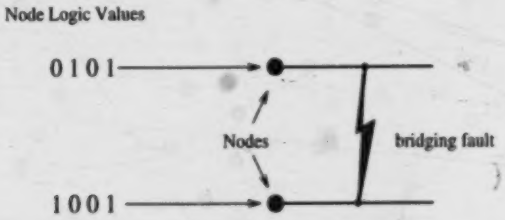


Figure 2.8: Activating a Bridging Fault

Figure 2.8 shows the four different pairs of signal values that can appear on the two circuit nodes. For the first input pair, both nodes are at logic 1. In this case, the bridging fault has no observable effect. Similarly, for the second input pair both nodes are at logic 0 so a bridge between the nodes will go unnoticed. However, for the third and fourth input pairs the nodes are at opposite logic values. The conflicting values indicate that the bridging fault is activated, and that the resulting logic value on both bridged nodes is uncertain. Moreover, since the circuit is attempting to drive either end of the fault to a different logic value, a high level of I_{DDQ} current will flow.

A non-feedback fault cannot be detected using the above method if the fault is between a pair of functionally equivalent nodes [20]. Two nodes that have identical logic values for every input vector applied to the circuit are defined as being *functionally equivalent.* Since these nodes can never hold opposite logic values, bridges between them are never activated and cannot be detected using I_{DDQ} testing.

Feedback Bridging Fault Detection

Since the two nodes in a feedback bridging fault have a logical dependence on each other, the activation of the fault can cause erratic circuit behaviour. Feedback bridges can cause oscillations and unwanted stable states. Consequently, feedback bridges do not always result in an easily measurable I_{DDQ} fault [14]. Figure 2.9 illustrates such a case.

The circuit in figure 2.9 has a bridging fault between node 5 and node 7. Consider the input vector (I1 I2 I3 I4) = (0 1 0 1). In the fault-free case, node 5 is set to 0, and

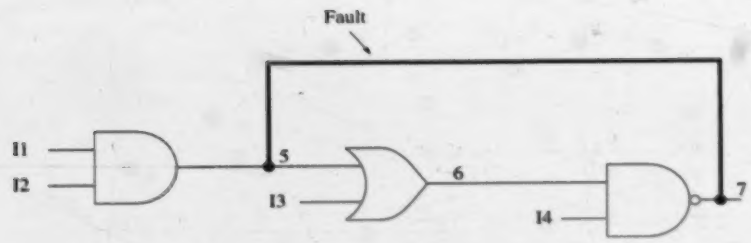


Figure 2.9: Feedback Bridging Fault

node 7 is set to 1. It can also be seen that the logic value at node 7 directly depends upon the logic value at node 5. If a bridge exists between nodes 5 and 7, the logic value at node 7 is fed back to node 5 and the voltage at node 5 will be increased. This change will then propagate through the OR gate to node 6, which will in turn affect the value at node 7. Since this propagated value is uncertain, it is not clear how node 7 will be affected. If node 7 is changed to a 0 by the uncertain value, there may no longer be opposite logic values on nodes 5 and 7. Consequently, there will be no low resistance current path from Vdd to GND, and no faulty I_{DDQ} will be observed.

If the input vector (I1 I2 I3 I4) is changed to (0 1 0 0), the logic value at node 7 no longer depends on the value at node 5. Since the two nodes are still forced to different logic values by this vector, (node 5 = 0 and node 7 = 1), a low resistance path from Vdd to GND will be set up in the circuit, and an I_{DDQ} fault can be detected.

This example shows that feedback bridging faults are not guaranteed to be I_{DDQ}-detectable using the same conditions as were used for non-feedback faults. To be certain that a feedback fault produces a high level of I_{DDQ}, the two nodes must not depend on one another during the application of a fault-activating test vector.

2.4.2 Limitations of I_{DDQ} Testing

Equations 2.5, 2.6, and 2.7 illustrate that the faulty level of I_{DDQ} produced from a bridging fault depends upon the magnitudes of the voltage levels in the circuit. Consequently, logic circuits designed to operate with 5 volt power supplies will produce higher levels of I_{DDQ} than logic circuits which use 3 volt power supplies.

When smaller voltages are used in digital circuits, the resulting smaller levels of I_{DDQ} current become more difficult to measure. The I_{DDQ} testing resolution is

subsequently reduced, and the difference between good and faulty levels of I_{DDQ} becomes smaller. Since low-power CMOS circuits are the current trend in digital design, I_{DDQ} testing will become more difficult to implement, and may not produce a high enough testing resolution to be a viable CMOS testing method. However, I_{DDQ} testing is still an active area of research and testing practice for the current CMOS technology available.

2.5 Fault Diagnosis in Digital Circuits

Fault diagnosis is an important part of circuit production as it can help to determine common or frequent errors in newly fabricated circuits. Once the common errors are known, they can be eliminated either by redesigning the circuit or by improving the fabrication process.

The normal diagnosis process for a digital circuit is as follows. Once a circuit has been declared faulty by some kind of testing method, the circuit is examined to determine which faults (specified by the fault model used during testing) are responsible for the malfunction. Once the error-causing faults are identified, diagnosis proceeds by physically examining the circuit in an attempt to find the actual defects.

Many of the previous works on fault diagnosis have used voltage-based schemes and the stuck-at fault model. More recently, some diagnostic methods have been proposed which target bridging faults as the most likely cause of circuit failures. This section reviews some previous diagnosis methods that have been proposed and highlights both positive and negative features of these methods.

Simulation-Based Diagnosis Methods

The conventional approach to fault diagnosis is the construction of a fault dictionary. A fault dictionary is constructed by simulating every modelled fault (usually single stuck-at faults) in the CUT and recording the results. The results for each modelled fault are then stored in the fault dictionary. By matching the output response of the CUT to those in the fault dictionary, the fault or fault equivalence class can be identified. The major deficiencies of a fault dictionary approach to diagnosis is the limiting fault model assumption, and the inability of the method to locate multiple

faults.

A number of proposals have been presented to diagnose multiple faults in digital circuits. Many proposals use simulation techniques to locate the faults or fault classes which can explain the faulty circuit behaviour. In [2], a multiple fault diagnosis method which uses an algorithm named the *Deduction Algorithm* is given. Given a test set, the algorithm uses back tracing techniques to deduce the logical values on the internal circuit lines based on the values on the primitive circuit outputs. By analysing these values, a list of potential stuck-at faults is produced which can account for the faulty behaviour at the circuit outputs. A similar diagnosis method is presented in [17]. This method uses pairs of input vectors to diagnose faulty nodes in the CUT. The diagnosis is performed in two steps. In the first step, the possible logic values that can appear on the circuit nodes are determined according to a specified fault model. In the second step, back tracing is done from the values on the primary circuit outputs to determine the actual value on the circuit nodes. In this step, it can be deduced which circuit lines are fault-free, so any faulty behaviour is isolated to the remaining circuit lines. As in [2], a list of potential faults or fault classes is produced which can account for the faulty circuit behaviour.

Signature Analysis Based Diagnosis

Signature analysis is an efficient data compaction technique that is used in built-in self-test (BIST) environments [9]. Many different diagnosis schemes exist to diagnose circuits where signature analysis is employed during testing. These schemes can be separated into three main categories which are specified by the type of analysis performed. The categories are fault dictionaries, intermediate signatures, and algebraic analysis.

Fault Dictionaries

Fault dictionaries in signature-based methods use look-up tables to compare the signatures from faulty circuits to the signature of a fault-free circuit. The look-up table is created by simulating faults or fault classes in a circuit, and recording the faulty signature that is produced. As with other fault dictionary approaches, multiple fault analysis is not possible due to the huge number of possible faults that can exist in the CUT. Another weakness of the fault dictionary approach is the possibility that

two or more faults will produce identical faulty signatures (aliasing). The possibility of aliasing reduces the diagnostic resolution achievable by this method.

Intermediate Signatures

Intermediate signature based methods partition long circuit responses into shorter blocks of data. When signature analysis is performed on the shorter data blocks, the resulting signature is compared to a pre-computed fault-free signature. A fault-free signature is required for every data block received from the CUT. After the entire set of test vectors has been applied to the CUT, only the data blocks which produce faulty signatures must be analysed further. This results in a large savings in the CPU time and memory required by the fault dictionary methods.

An intermediate signature based diagnosis method is presented in [44]. In this method, signatures are collected in a multiple-input shift register (MISR) after every 256 test patterns. If a faulty signature is found, the block of faulty data is stored. Diagnosis using fault dictionaries and fault simulation is then performed using data from the failing blocks. This method provides a high degree of diagnostic resolution, but it also requires a large amount of memory to store the failing blocks of data and the intermediate signatures.

Algebraic Analysis

Algebraic analysis diagnostic methods use the mathematical relationships between faulty output data from a CUT and the corresponding signatures to identify the failing outputs from the CUT during a test session. In [28], the "error signature" of a linear feedback shift register (LFSR) [10, page 145] is analysed to determine which output from the CUT is responsible for producing a faulty signature. If the output sequence from the CUT is less than $2^n - 1$ bits long, and if the output sequence has fewer than three errors, the method in [28] can determine which bits in the sequence are faulty. A similar method for diagnosing output sequences when MISRs are used is presented in [16]. This method uses a matrix formulation to describe the output data and perform the diagnosis.

A similar method which employs cyclic registers is given in [36]. By employing two cycling registers, the maximum testable length of the output sequence from the CUT is increased to the least common multiple of the degrees of the cycling registers. In [39], a method is presented which uses two LFSRs to perform the diagnosis. Here the

length of the output sequence that can be analysed is increased to the least common multiple of the orders of the polynomials which describe the LFSRs.

In all of the above algebraic methods, the maximum length of the output sequence is much shorter than those produced from practical circuits. This fact, along with the fact that the methods are only successful at diagnosing output sequences which have very few errors, implies that these methods are not useful for diagnosing large VLSI circuits.

An analysis on the diagnostic properties of LFSRs is given in [33]. The main goal in [33] is to develop a formula that will determine the fraction of circuit faults that will be uniquely diagnosed (produce a unique signature) for a given circuit size and LFSR length. Although mapping specific faults to specific signatures may provide some diagnostic information, the problem of aliasing and the large number of faults that can exist in VLSI circuits make diagnostic schemes based solely on this method infeasible.

In [5] a diagnosis method is presented for multiple output circuits that are tested using pseudorandom test vectors as inputs. This method uses detection probabilities [5] which are precomputed for each fault to help diagnose faults quickly during signature compaction. A special type of MISR called a MINSR [5], is used to perform the compaction while the quotient stream output from the MINSR is analysed to provide an estimate for the detection probability. To differentiate between faults that have similar detection probabilities, post-test simulation is performed.

Diagnosis in a STUMPS Environment

Many sequential circuits are designed using scan-based flip-flops to increase the testability of the circuit. Because input data can be scanned into a sequential circuit through a scan chain created by connecting the scan flip-flops together, a sequential circuit can be converted into a combinational circuit for testing purposes. Combinational circuits are far easier to test than sequential circuits.

A scan-based testing architecture that is widely used is the Self-Test Using MISR/ Parallel SRSG[1] (STUMPS) testing architecture [9]. STUMPS is a multichip testing architecture used to reduce the testing time at the board level. A test chip is used

[1]SRSG stands for shift-register sequence generator.

to both provide test patterns to the chips and compact the output test responses in parallel from the chips. The memory elements in each chip are configured into scan chains prior to a test. During a test, data is scanned into the chains using a test clock. Once each memory element is loaded, the system clock is asserted once to capture the circuit responses back into the scan chains. The circuit responses are then scanned out of the chains and compacted by the test chip. The resulting signature produced by the compaction is subsequently compared to a fault-free signature to see if an error is detected.

STUMPS can also be used to test circuits at the IC level by partitioning the circuit into multiple scan chains. In this case, the test pattern generator and data compactor can be built-in test resources, instead of existing on a separate test chip.

In a STUMPS-based testing architecture, it is important for diagnostic purposes to discover which of the flip-flops in the circuit have at one time collected faulty data from the combinational sections of the circuit. This information is analogous to determining which primary output has at one time had a faulty logic level propagated to it. The following paragraphs present two methods used to diagnose the "fault-capturing" flip-flops in a STUMPS testing architecture.

The diagnosis method in [45] uses coding theory and signature analysis to identify the fault-capturing flip-flops in a STUMPS testing architecture. By using a special programmable MISR (PMISR) to compute signatures based upon Reed-Solomon codes [23], a set of equations are produced which when solved, identify the fault-capturing flip-flop frames [45]. Each chain in the STUMPS architecture can then be retested in order to locate the actual fault-capturing flip-flops.

While this diagnosis method provides good diagnostic resolution, its implementation can be difficult. For large circuits having many faults, it can take a large amount of computational power to solve the system of equations that is produced. Moreover, the diagnosis requires a long time to run, and the number of faulty scan-frames that the method can handle is limited.

The method in [40] also identifies the fault-capturing flip-flops in a STUMPS-based testing architecture. This method uses signature analysis to systematically find the faulty chains and flip-flops during diagnosis using a divide-and-conquer strategy. For every flip-flop in the system, a corresponding signature can be calculated that will

inform the user whether that flip-flop has captured faulty data at one time during the diagnosis run.

This diagnosis method provides as good a diagnostic resolution as that in [45] and does not require that any equations be solved. It also has no limitations on the number of faults that can exist in the circuit. Some possible limitations of the method are the amount of memory required to store the fault-free signatures, and the amount of time required to perform a diagnostic run.

Bridging Fault Diagnosis

In recent years, research has been done in the area of bridging fault diagnosis. Most of the work done has dealt with diagnosing faults in CMOS circuits. The following paragraphs discuss some of the diagnosis methods that have been recently developed. The methods fall into two basic categories: fault dictionary methods and algorithmic methods.

Fault Dictionary Methods

Stuck-at fault dictionaries are used to diagnose CMOS bridging faults in [31]. The diagnosis method uses information from a stuck-at fault dictionary, and the relationships between stuck-at faults and low-resistance bridging faults to perform a diagnosis. It is claimed in [31] that for tests performed on benchmark circuits, over 92% of bridging faults in the circuit can be diagnosed correctly.

Although the method in [31] produces a high diagnostic resolution it has some drawbacks. The most serious drawback is the time and memory required to construct and store a complete stuck-at fault dictionary. For large circuits, the amount of memory required may make the construction of a dictionary infeasible. A second potential drawback is the way that the method handles multiple bridging faults. If the errors that propagate to the circuit output due to multiple bridging faults mask each other, the diagnosis may be misleading. Moreover, if the bridging faults in the circuit have non-zero values of resistance, they may not propagate logic errors to the outputs at all. The diagnosis of such faults is impossible using this method.

Fault dictionaries are also used for CMOS bridging fault diagnosis in [22]. In [22], a mixed-mode (switch-level and gate-level) simulator is used to build three different types of fault dictionaries that are used in the diagnosis. The first dictionary is known

as the *complete fault dictionary.* For this dictionary, a list of detected faults at each primary output is constructed for each input vector. The second dictionary is called the *pass/fail dictionary.* It associates the list of detected faults with each input vector instead of with each primary output at each input vector. The third dictionary is called the *count dictionary.* It contains the maximum and the minimum detections of each fault.

It is shown in [22] that the complete fault dictionary provides a very high diagnostic resolution. However, a great deal of computer memory is required to implement such a dictionary, even when small circuits are considered. It is also stated that the other two dictionary types do not provide as high a quality of resolution, but can produce acceptable resolution if I_{DDQ} information is incorporated along with the regular voltage information.

In [6], a diagnosis method which uses both I_{DDQ} test information as well as voltage information to locate faults is presented. The method uses a transistor leakage (transistor bridging) fault model to model circuit defects, although the method can also be used with other fault models [7]. The main idea of the method is to apply a test set to the CUT and monitor the I_{DDQ} levels. Every time a faulty level of I_{DDQ} is measured, the primary circuit outputs are stored, as is the sequence number of the input test pattern. Therefore, both the input vector and the output response to the circuit will be stored when an I_{DDQ} fault is activated. After a pre-specified number of faults have been found (32 faults are used in [6]) fault simulation is performed for each fault in a fault list in order to determine which faults may be responsible for the faulty behaviour.

Since this method uses I_{DDQ} test results to aid in diagnosis, it performs well at diagnosing bridging faults. However for large circuits, the time and memory problems associated with constructing fault dictionaries, storing output responses, and performing fault simulation still exist. Therefore, this method can be expensive to implement.

Algorithmic Methods

Many papers have been published by Chakravarty regarding bridging fault diagnosis. These papers use special algorithms to locate bridges based upon different test results. In [14] an algorithm is presented to diagnose two-node bridging faults in

CMOS circuits based upon I_{DDQ} test information. The algorithm employs a single fault assumption, and performs diagnosis at the gate level. Although the algorithm is very efficient, it is limited by the single fault assumption. Moreover, when feedback bridging faults are considered, the algorithm is more pessimistic than it needs to be. This leads to a lower diagnostic resolution than is possible.

In [13] an algorithm is given which uses the wired-OR fault model to describe the function of bridges in a circuit. When an input vector is applied to the circuit, a set of faults is computed that are detected by that vector. A collection of these fault sets comprise a fault dictionary. The algorithm works by analysing a set of input vectors and a particular test result. The output of the algorithm is a list of the possible faults that may be responsible for the test result being observed.

The algorithm in [13] requires less memory storage than the diagnosis method given in [31]. The limitation of the method is its dependence on the wired-OR fault model to describe bridges.

In [15], Chakravarty provides an updated version of the algorithm in [14]. Assuming that it is known which input test vectors activate I_{DDQ} faults, the algorithm is able to diagnose both internal gate bridges, and external gate bridges. The limitation of the diagnosis algorithm, as in [14], is that it uses the single fault assumption.

The following chapter presents two new bridging fault diagnosis algorithms: a single fault algorithm which improves on the algorithm in [14], and a multiple fault algorithm which can diagnose circuits containing more than one fault.

Chapter 3

New Algorithms

In this chapter, two new diagnosis algorithms for gate-level bridging fault diagnosis in CMOS circuits are presented. One algorithm employs the conventional single fault assumption, while the second algorithm is applicable to circuits containing multiple bridging faults. The terminology used to describe the algorithms is defined, and examples to illustrate the operations of both algorithms are presented. The computational complexity of the algorithms is given. Finally, subsequent diagnostic methods which use the information provided by the new algorithms are discussed.

3.1 Testing and Diagnosis Process

The diagnosis algorithms are part of a complete bridging fault testing and diagnosis process. The algorithms incorporate the results of an I_{DDQ} test session on a CUT along with the results of logic simulations to determine the single bridging faults and multiple bridging faults that can cause a circuit to produce test results identical to those observed from the CUT. The use of gate-level logic simulation allows diagnosis to be performed much more quickly than can be done when circuit-level simulation is used to determine the same information.

It is assumed that some type of I_{DDQ} testing (off-chip or built-in) can be easily performed on the CUT. However, the actual I_{DDQ} testing implementation is beyond the scope of this thesis.

The testing and diagnosis process proceeds as follows.

1. Bridging fault testing:

Step 1: Generate a set of I_{DDQ} test vectors.

Step 2: Perform an I_{DDQ} test on the CUT using the vectors in the test set. If the CUT fails the test during the application of the n-th vector, the number n is recorded.

If at least one test vector causes an I_{DDQ} fault to be detected in the CUT, the CUT is declared *faulty* and diagnosis using the new algorithms is performed.

2. Bridging fault diagnosis:

Step 1: Generate an initial set of possible faults. This set can be obtained by extracting information from the layout of the CUT and determining which nets are routed close enough together so that bridges may exist between them. If no layout information about the CUT is available, an exhaustive set of bridging faults can be used.

Step 2: Perform logic simulations on a copy of the CUT (gate level) using the same set of test vectors used during the I_{DDQ} testing. The simulations are used to obtain logic information for each node in the CUT. These simulations can be performed either prior to or during the execution of the algorithm(s).

Step 3: Invoke the single fault diagnosis algorithm using the simulation information, the I_{DDQ} test results, and the initial fault set as inputs. The algorithm produces two fault sets which contain the diagnosed faults.

Step 4: If the single fault diagnosis algorithm does not identify any single fault which is responsible for the faulty I_{DDQ} results from the CUT, invoke the multiple fault algorithm. Initially, it should be assumed that the CUT contains double faults, as they are the most probable type of multiple fault. If the diagnostic information produced using the double fault assumption is still not sufficient, triple, quadruple, or other multiple fault assumptions can be used in turn.

In summary, the new diagnosis algorithms produce sets of faults which can be responsible for the faulty I_{DDQ} results observed during the testing process (i.e. the fault equivalence classes). To determine the actual bridging fault(s) from these sets that is

in the faulty CUT, further diagnostic methods such as circuit-level fault simulation or electron beam testing [46] must be employed.

To use circuit-level fault simulation for diagnostic purposes, the following steps can be performed. The netlist of the CUT is input to a circuit-level fault simulator. Bridging faults diagnosed by the algorithms are then injected into the netlist. The simulator applies a set of input vectors to the netlist and the resulting output logic levels are recorded. The same input vectors are then used to stimulate the CUT, and the output logic levels from the CUT are recorded. This set of test vectors may or may not be the same set of vectors used by the diagnosis algorithms. Since circuit-level fault simulation can take a long time, a small set of test vectors is normally used.

If the outputs from the simulated netlist are different than the actual outputs from the CUT, it is assumed that the fault injected into the netlist is not present in the CUT. If the two sets of outputs match exactly, the injected fault may be present in the CUT. The above process is repeated until all the faults diagnosed by the algorithms have been tested.

If the results provided from circuit-level fault simulation are not sufficient, or if the exact nature of the fault at the layout level must be investigated, physical diagnosis of the CUT can be performed using an electron beam testing system. An electron beam tester can physically trace the CUT in order to find the circuit defects responsible for faulty circuit behaviour. Although an electron beam tester provides the highest possible resolution of any diagnostic method, it requires a great amount of time to perform a diagnosis. Consequently, electron beam testing is normally performed after the diagnostic information from other methods has been collected.

3.2 Fault Assumptions

Gate-level bridges are the most common type of bridge in CMOS circuits designed using standard cell libraries [34]. It is assumed for this thesis that bridging faults occur at the gate level.

Both feedback bridging faults and non-feedback bridging faults are considered during diagnosis. Due to the unpredictable effects produced by feedback bridging

faults (see chapter 2 page 12) the assumptions made regarding the I_{DDQ} testability of feedback faults in the CUT will affect the resolution achieved by the diagnosis algorithms.

As is discussed in chapter 2, a test vector which attempts to force the nodes in a feedback fault to opposite logic levels does not necessarily produce a faulty level of I_{DDQ}. If one of the two nodes in the fault directly influences the logic level on the second node when a specific test vector is applied, the current path between Vdd and GND may be eliminated due to the logical dependence between the nodes. However, without the use of a circuit-level simulator, the exact behaviour of feedback faults to each test vector is not readily known. Without this knowledge, assumptions must be made regarding whether feedback faults will always produce faulty I_{DDQ} levels when they are activated.

For the new diagnosis algorithms, a relatively conservative approach is taken. It is assumed that if the signal on one node involved in a feedback fault is responsible for determining the signal on the second node for a particular test vector, it cannot be determined whether the fault will produce a faulty level of I_{DDQ} during the application of that test vector. Therefore, when a test vector produces no faulty I_{DDQ} levels in the CUT, feedback faults can only be eliminated when the nodes in the fault are logically independent.

3.3 Definitions and Declarations

This section provides some definitions and terminology that are used by the diagnosis algorithms.

Let T be a set of test vectors used to stimulate a circuit while I_{DDQ} testing is performed on that circuit. t is the total number of test vectors in T. Let X_{T_i} be the logic value of a node X when test vector T_i is applied to the CUT.

The graph in figure 3.1 illustrates the results of a sample I_{DDQ} test session where $t = 10$. For each test vector, there is a short transient component of the circuit's current and a longer steady-state component. I_{DDQ} testing is performed during the steady-state. It can be seen from figure 3.1 that 4 of the 10 test vectors fail the I_{DDQ} test, since they produce a steady-state current which is higher than the specified

threshold current.

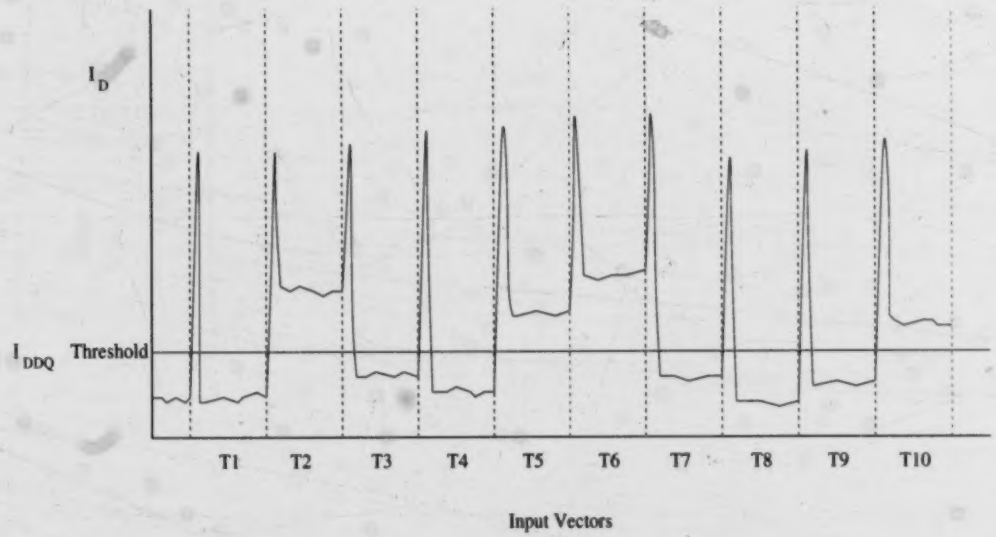


Figure 3.1: Circuit Current During I_{DDQ} Test Session

Definition 3.1 If the two nodes at either end of a fault are set to opposite logic values by a particular test vector, the fault is said to be *activated* by that vector.

For the example in figure 3.1, faults are activated during the application of test vectors 2, 5, 6, and 10.

Definition 3.2 The *activation pattern (AP)* is the set of integers corresponding to the indices of the test vectors which produce faulty I_{DDQ} test results. For each T_i in T $(1 \leq i \leq t)$ that activates an I_{DDQ} fault, the integer $i \in AP$.

AP_{CUT} is the AP containing the I_{DDQ} test result information obtained from the CUT. For the example in figure 3.1, $AP_{CUT} = \{2,5,6,10\}$.

$AP_{SIM} = \{AP_{SIM_1}, AP_{SIM_2}, ..., AP_{SIM_{TOTI}}\}$ is the class of APs formed during logic simulation of the CUT, where $TOTI$ is the total number of faults in the CUT. Each set in AP_{SIM} contains the set of integers which describe the I_{DDQ} test results for a simulated copy of the CUT, where the simulated copy contains a single bridging fault.

Definition 3.3 The *activation frequency (AF)* is the number of test vectors in a test set T that produce a faulty level of I_{DDQ}.

For the test set in the example in figure 3.1, $AF = 4$.

Definition 3.4 A feedback bridging fault is said to be *sensitized* by the current test vector if the logic value of one of the nodes in the fault depends directly on the logic value on the other node.

To determine whether a feedback fault is sensitized, the node in the fault closest to a primary input is complemented. If the other node in the fault is affected by the change to the first node (either permanently or temporarily), the fault is sensitized by the current test vector.

Fault Set Declarations

This subsection declares the different fault sets which are either used or produced by the new diagnosis algorithms.

1. A set of faults, F, is denoted as $\{f_1, f_2, \ldots, f_{TOTI}\}$, where $TOTI$ is the number of faults in F.

2. F_I is the initial set of bridging faults passed to either algorithm.

   F_I is divided into two sets, F_F and F_{NF}. F_F is the set of feedback faults from F_I, while F_{NF} is the set of non-feedback faults from F_I.

3. F_L is the set of bridging faults which are identified by either diagnosis algorithm as faults which may be responsible for all faulty I_{DDQ} levels recorded in the CUT. When the multiple fault algorithm is invoked, F_L also contains faults which are known to be responsible for the faulty I_{DDQ} level observed for one particular test vector. For a single fault f_j, $f_j \in F_L$ if $(AP_{SIM_j} = AP_{CUT})$ OR $(AP_{SIM_j} \backslash AP_{SIM_i}) \neq \emptyset$, where $1 \leq i, j \leq TOTI$, $i \neq j$, and $(AP_{SIM_j} \backslash AP_{SIM_i}) \in AP_{CUT}$.

4. F_P is the set of bridging faults which are identified by the multiple fault algorithm as faults which may be present in the CUT, but cannot be solely responsible for all of the I_{DDQ} faults observed in the CUT; that is, $f_j \in F_P$ if $(AF_j < AF_{CUT})$ AND $(AP_{SIM_j} \setminus AP_{CUT}) = \emptyset$.

5. F_S is the set of feedback bridging faults which cannot be eliminated by the diagnosis algorithms due to the fact that they are sensitized for specific test vectors, that is, $f_j \in F_S$ if $(AP_{SIM_j} \setminus AP_{CUT}) \neq \emptyset$ AND f_j is sensitized for all T_i, where $i \in \{AP_{SIM_j} \setminus AP_{CUT}\}$.

6. F_{SE} is the subset of faults from set F_S which may be solely responsible for all of the I_{DDQ} failures recorded from the CUT, that is, $f_j \in F_{SE}$ if $(AP_{CUT} \setminus AP_{SIM_j}) \neq \emptyset$. The single fault algorithm diagnoses sensitized faults to F_{SE} only.

7. F_U is the set of bridging faults which are not tested by a specific test set T, that is, $f_j \in F_U$ if $AF_j = 0$. F_U is produced only by the multiple fault algorithm.

8. $F_N(x)$ is the set of multiple bridging faults where each multiple fault is a combination of x single faults and x is an integer. $F_N(x)$ contains all of the single fault combinations from sets F_P and $(F_S \setminus F_{SE})$ which can combine to produce $AP_{SIM_n} = AP_{CUT}$, where AP_{SIM_n} is the AP of one multiple fault.

   $F_N(x)$ is split into to subsets. F_{NP} is the set of faults in $F_N(x)$ which are made up of single faults from set F_P only. F_{NS} is the set of faults in $F_N(x)$ which contain at least one single fault from $(F_S \setminus F_{SE})$.

3.4 Single Fault Diagnosis

Many testing and diagnosis schemes assume that only one fault is present in a circuit during a test session. This is known as the single fault assumption [3, page 94]. This assumption is justified when tests are performed frequently enough so that the probability of more than one fault developing between test sessions is sufficiently small.

This section presents a new diagnosis algorithm for circuits containing single bridging faults. The new algorithm is an improvement on the algorithm presented in [14] as the new algorithm handles feedback bridging faults in a more optimistic manner. As a result, a higher diagnostic resolution is obtained. This leads to savings in the time and computational power required during subsequent diagnosis processes that are performed using the information provided by our diagnosis algorithm.

In the new single fault algorithm, logic simulation results are used to analyse each fault in the initial fault set (F_I) passed to the algorithm. If the simulation results indicate that a particular fault, f_j, will produce an AP_{SIM_j} which is identical to AP_{CUT}, the fault is placed into F_L. The algorithm also places feedback faults into F_S if the fault is sensitized by every test vector T_i where $i \in AP_{SIM_j}$ and $i \notin AP_{CUT}$. The outputs of the algorithm are the faults in both F_L and F_S.

3.4.1 Single Fault Algorithm

The inputs to the algorithm are AP_{CUT}, and F_I. Prior to execution of the algorithm, each fault in F_I is classified as either a feedback fault, or a non-feedback fault, and is placed into fault sets F_F and F_{NF} accordingly. The algorithm proceeds as follows (a flowchart of the algorithm can be found in figure A.1 of appendix A).

```
procedure SINGLE_FAULT_ALG()
begin
F_L0 = F_I                          /* Initial classification of fault */
F_S0 = ∅                            /* lists F_L and F_S. */

for i = 1 to t                      /* Analysis is over all t test vectors */
   for j = 1 to TOTI                /* Analyse every fault in the fault list */
                                    /* that has not been eliminated */
      actflag = ACTIVATION(f_j)     /* Determine if f_j is activated by the */
                                    /* current test vector */

      if (i ∈ AP_CUT) then          /* If an I_DDQ fault is detected in the */
                                    /* CUT by T_i */
         if (actflag = false) then  /* Eliminate fault f_j if it is not activated */
                                    /* by T_i */
            eliminate f_j
         else                       /* If fault f_j is activated by T_i */
                                    /* keep it in the same fault set as before */
            if (f_j ∈ F_Li-1) then
               f_j ∈ F_Li
            else if (f_j ∈ F_Si-1) then
               f_j ∈ F_Si
            end if
         end if

      else                          /* If no I_DDQ fault is detected in the */
                                    /* CUT by T_i */
         if (actflag = false) then  /* If fault f_j is not activated by T_i */
```

```
                                        /* keep it in the same fault list as before */
            if (f_j ∈ F_{L_{i-1}}) then
                f_j ∈ F_{L_i}
            else if (f_j ∈ F_{S_{i-1}}) then
                f_j ∈ F_{S_i}
            end if
        else                            /* If fault f_j is activated by T_i */
            if (f_j ∈ F_L) OR (f_j ∈ F_S) then
                if (f_j ∈ F_NF) then    /* If f_j is a non-feedback fault, */
                                        /* eliminate it */
                    eliminate f_j
                else                    /* If f_j is a feedback fault */
                    if (f_j is sensitized) then
                                        /* If f_j is sensitized, place it in F_S */
                        f_j ∈ F_{S_i}
                    else                /* If f_j is not sensitized, eliminate it */
                        eliminate f_j
                    end if
                end if
            end if
        end if
    end for
end for
end procedure                           /* End SINGLE_FAULT_ALG */


procedure ACTIVATION(f_j)               /* Determines whether fault f_j is */
                                        /* activated by test vector T_i */


begin
if (X_{Ti_j} = Y_{Ti_j}) then           /* If the logic values on the nodes are */
    flag = false                        /* equal, flag = false (not activated) */
else                                    /* If the logic values are not equal, */
    flag = true                         /* flag = true (activated) */
end if
return flag                             /* Return flag */
end procedure                           /* End ACTIVATION */
```

3.4.2 Single Fault Diagnosis Example

Figure 3.2 shows a small combinational digital circuit. To help illustrate the single fault diagnosis algorithm, an example diagnosis run will be performed on this circuit.

The circuit in figure 3.2 has eight different nodes. Therefore, there is a total of $C_2^8 = 8!/(2! \times 6!) = 28$ two-node bridging faults. Each of these faults is represented

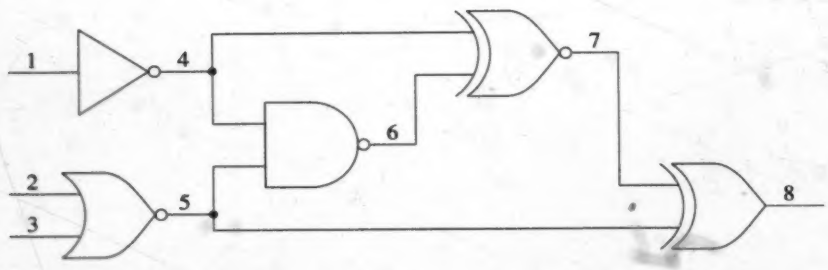


Figure 3.2: CUT for Example Diagnosis Run

as a node pair. Table 3.1 is divided into two halves. The left half shows each different node pair and indicates whether the node pair is a feedback (F) or a non-feedback (N) fault. The right half of table 3.1 gives the corresponding fault number for each node pair. For example, node pair 1-2 is labelled as fault f_1, and node pair 6-8 is labelled as fault f_{27}. The fault related information for the circuit in figure 3.2 is summarized in table 3.2.

Nodes	Nodes								Corresponding Faults								
	1	2	3	4	5	6	7	8		1	2	3	4	5	6	7	8
1	-	N	N	F	N	F	F	F	1	-	f_1	f_2	f_3	f_4	f_5	f_6	f_7
2	-	-	N	N	F	F	F	F	2	-	-	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}
3	-	-	-	N	F	F	F	F	3	-	-	-	f_{14}	f_{15}	f_{16}	f_{17}	f_{18}
4	-	-	-	-	N	F	F	F	4	-	-	-	-	f_{19}	f_{20}	f_{21}	f_{22}
5	-	-	-	-	-	F	F	F	5	-	-	-	-	-	f_{23}	f_{24}	f_{25}
6	-	-	-	-	-	-	F	F	6	-	-	-	-	-	-	f_{26}	f_{27}
7	-	-	-	-	-	-	-	F	7	-	-	-	-	-	-	-	f_{28}
8	-	-	-	-	-	-	-	-	8	-	-	-	-	-	-	-	-

Table 3.1: Nodes and Faults for Example Circuit

Gates	Nodes	Bridging Faults	Feedback Faults	Non-Feedback Faults
5	8	28	21	7

Table 3.2: Circuit Information for Diagnosis Example

Since the CUT has only 3 inputs, an exhaustive test is used to achieve the maximum diagnostic resolution. In a practical circuit, the number of inputs is normally too large to allow for the use of an exhaustive test. In such a case, a much smaller test set, such as a set of pseudorandom test vectors, or a set of specially generated

I_{DDQ} test vectors is used instead of an exhaustive set. Table 3.3 shows all of the test vectors used, the order in which they are applied to the CUT , the I_{DDQ} test results for each vector, and the fault-free logic values on each circuit node during the test.

Input Vector		Node Logic Values								I_{DDQ} Test Results
i		1	2	3	4	5	6	7	8	
1	000	0	0	0	1	1	0	0	1	Pass
2	001	0	0	1	1	0	1	1	1	Fail
3	010	0	1	0	1	0	1	1	1	Fail
4	011	0	1	1	1	0	1	1	1	Fail
5	100	1	0	0	0	1	1	0	1	Fail
6	101	1	0	1	0	0	1	0	0	Pass
7	110	1	1	0	0	0	1	0	0	Pass
8	111	1	1	1	0	0	1	0	0	Pass

Table 3.3: Logic Simulation and I_{DDQ} Test Results for Diagnosis Example

From the I_{DDQ} test results, $AP_{CUT} = \{2, 3, 4, 5\}$. The algorithm requires AP_{CUT} and F_I (which is divided into F_F and F_{NF}) as input. The algorithm starts by initially classifying the faults from F_I into sets F_{L_0} and F_{S_0} as follows:

$F_{L_0} = \{f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}, f_{11}, f_{12}, f_{13}, f_{14}, f_{15}, f_{16}, f_{17}, f_{18}, f_{19}, f_{20}, f_{21}, f_{22}, f_{23}, f_{24}, f_{25}, f_{26}, f_{27}, f_{28}\}$

$F_{S_0} = \emptyset$

Faults are either eliminated or repositioned into F_L or F_S by analysing the I_{DDQ} test result and the logic simulation values for each test vector ($1 \leq i \leq 8$).

$i = 1$:

Since $1 \notin AP_{CUT}$, all activated non-feedback faults (f_4, f_9, and f_{14}), and any activated feedback faults which are not sensitized (none), are eliminated. All activated feedback faults which are sensitized (f_3, f_7, f_{10}, f_{13}, f_{15}, f_{18}, f_{20}, f_{21}, f_{23}, f_{24}, f_{27}, and f_{28}) are placed in F_{S_1}. The remaining faults (f_1, f_2, f_5, f_6, f_8, f_{11}, f_{12}, f_{16}, f_{17}, f_{19}, f_{22}, f_{25}, and f_{26}) are placed in F_{L_1}. After all the information from test vector 1 has been analysed, 3 faults are eliminated and F_{L_1} and F_{S_1} are as follows:

$F_{L_1} = \{f_1, f_2, f_5, f_6, f_8, f_{11}, f_{12}, f_{16}, f_{17}, f_{19}, f_{22}, f_{25}, f_{26}\}$

$F_{S_1} = \{f_3, f_7, f_{10}, f_{13}, f_{15}, f_{18}, f_{20}, f_{21}, f_{23}, f_{24}, f_{27}, f_{28}\}$

$i = 2$:

Since $2 \in AP_{CUT}$, all non-activated faults (f_1, f_{10}, f_{16}, f_{17}, f_{18}, f_{20}, f_{21}, f_{22}, f_{26}, f_{27}, and f_{28}) are eliminated. All activated faults from F_{L_1} (f_2, f_5, f_6, f_8, f_{11}, f_{12}, f_{19}, and f_{25}) are placed in F_{L_2}, and all activated faults from F_{S_1} (f_3, f_7, f_{13}, f_{15}, f_{23}, and f_{24}) are placed in F_{S_2}. After the information from test vector 2 is analysed, F_{L_2} and F_{S_2} are as follows:

$F_{L_2} = \{f_2, f_5, f_6, f_8, f_{11}, f_{12}, f_{19}, f_{25}\}$

$F_{S_2} = \{f_3, f_7, f_{13}, f_{15}, f_{23}, f_{24}\}$

$i = 3$:

As was the case for test vector 2, test vector 3 activates an I_{DDQ} fault ($3 \in AP_{CUT}$). Therefore, all non-activated faults (f_2, f_{11}, f_{12}, f_{13} and f_{15}) are eliminated. The remaining faults are placed in F_{L_3} and F_{S_3}:

$F_{L_3} = \{f_5, f_6, f_8, f_{19}, f_{25}\}$

$F_{S_3} = \{f_3, f_7, f_{23}, f_{24}\}$

$i = 4$:

Since $4 \in AP_{CUT}$, all non-activated faults (f_8) are eliminated, while all activated faults are placed in F_{L_4} and F_{S_4}:

$F_{L_4} = \{f_5, f_6, f_{19}, f_{25}\}$

$F_{S_4} = \{f_3, f_7, f_{23}, f_{24}\}$

$i = 5$:

Since $5 \in AP_{CUT}$, all non-activated faults (f_5, f_7, f_{23}, and f_{25}) are eliminated while the rest are placed in F_{L_5} and F_{S_5}:

$F_{L_5} = \{f_6, f_{19}\}$

$F_{S_5} = \{f_3, f_{24}\}$

$i = 6$:

Since $6 \notin AP_{CUT}$, all activated and sensitized feedback faults (f_3, f_6, and f_{24}) are placed into F_{S_6}. f_{19} is a non-activated, non-feedback fault, so it is placed in F_{L_6}:

$F_{L_6} = \{f_{19}\}$

$F_{S_6} = \{f_3, f_6, f_{24}\}$

$i = 7$ and $i = 8$:

Examining the simulation results and checking the feedback paths for the faults in

F_{L_6} and F_{S_6} for both test vectors 7 and 8, shows that no further fault elimination or movement can be done using information from the last two test vectors. Therefore, the final output provided by the algorithm is:

$F_{L_8} = \{f_{19}\}$

$F_{S_8} = \{f_3, f_6, f_{24}\}$

The four faults left in the last F_L and F_S are the bridging faults which may be responsible for the faulty levels of I_{DDQ} found during the testing of the CUT. Since $AP_{SIM_{19}} = AP_{CUT}$, it is more likely that f_{19} and not one of the faults in F_S (f_3, f_6, and f_{24}) is the fault present in the CUT. In order to determine for certain which of the four faults is actually in the CUT, further diagnosis using other methods must be done. For example, logic information from the CUT outputs produced during fault simulation may provide enough information to determine which of the four bridging faults is actually in the CUT. Experimental study on the performance of the single fault algorithm is given in Chapter 4.

3.5 Multiple Fault Diagnosis

The single fault algorithm is used for circuits which are assumed to contain only one bridging fault at any one time. If more than one fault exists in the CUT, the single fault algorithm may misdiagnose one or all of the faults. This section introduces a new multiple fault diagnosis algorithm to be used when the single fault algorithm and successive diagnostic procedures fail to locate any defects in the CUT.

The multiple fault algorithm consists of 4 main procedures. The first procedure, called CLASSIFY, is used to analyse simulation results from the CUT and either categorize the faults into fault sets F_L, F_P, and F_S, or eliminate them. The second procedure, called UNDETECTED, places all faults which are not testable by the current test set into fault set F_U. The third procedure, called AP_UNIQUE, analyses the faults in F_P and F_S to determine if any single fault in these sets can be solely responsible for producing a faulty I_{DDQ} level in the CUT. If such a fault is found, it is moved into F_L. The last procedure, called NTUPLE, creates the multiple fault list F_N. F_N shows which faults in F_P and F_S can produce AP_{CUT} if they appear in the CUT simultaneously.

3.5.1 Multiple Fault Algorithm

The inputs required by the multiple fault algorithm are AP_{CUT}, F_I (divided into F_F and F_{NF}), and the degree of multiple faults that will be considered (i.e. double faults, triple faults etc.). The outputs of the algorithm are F_L, F_P, F_S, F_N (which can be divided into F_{NP} and F_{NS}), and F_U. The multiple fault algorithm proceeds as follows (flowcharts for the algorithm are given in figures A.2 to A.5 in appendix A).

```
procedure MULTI_FAULT_ALG()
CLASSIFY()                          /* Call CLASSIFY procedure to */
                                    /* construct F_L, F_P, and F_S */
UNDETECTED()                        /* Call UNDETECTED procedure to */
                                    /* construct F_U */
if F_L = ∅ then
   AP_UNIQUE()                      /* Call AP_UNIQUE procedure to */
                                    /* analyse F_P and F_S */
end if
NTUPLE(x)                           /* Call NTUPLE procedure to */
                                    /* create F_N(x) */
end procedure                       /* End MULTI_FAULT_ALG */


procedure CLASSIFY()                /* CLASSIFY constructs fault sets */
                                    /* F_L, F_P, and F_S */

F_L0 = F_I                          /* Initial classification of F_L, F_P, and F_S */
F_P0 = F_S0 = ∅

for i = 1 to t                      /* Analysis is over all t test vectors */
   for j = 1 to TOTI                /* Analyse all faults that have not been */
                                    /* eliminated */
      actflag = ACTIVATION(f_j)     /* Determine if f_j is activated by the */
                                    /* current test vector */

      if (i ∈ AP_CUT) then          /* If an I_DDQ fault is detected in the */
                                    /* CUT by T_i */
         if (actflag = false) then  /* If f_j is not activated by the current */
                                    /* test vector */
            if (f_j ∈ F_Li-1 OR f_j ∈ F_Pi-1) then
                                    /* If f_j is in F_L or F_P, place it in F_P */
               f_j ∈ F_Pi
            else                    /* Keep f_j in F_S */
               f_j ∈ F_Si
            end if
```

```
        else                                    /* If f_j is activated by T_i */
                                                /* keep f_j in the same fault list it was in */
                                                /* during the last loop. */
            if (f_j ∈ F_{L_{i-1}}) then
                f_j ∈ F_{L_i}
            else if (f_j ∈ F_{P_{i-1}}) then
                f_j ∈ F_{P_i}
            else if (f_j ∈ F_{S_{i-1}}) then
                f_j ∈ F_{S_i}
            end if
        end if

    else                                        /* If no I_DDQ fault is detected in the */
                                                /* CUT by T_i */
        if (actflag = false) then               /* If fault f_j in not activated by T_i */
                                                /* keep f_j in the same fault set as before */
            if (f_j ∈ F_{L_{i-1}}) then
                f_j ∈ F_{L_i}
            else if (f_j ∈ F_{P_{i-1}}) then
                f_j ∈ F_{P_i}
            else if (f_j ∈ F_{S_{i-1}}) then
                f_j ∈ F_{S_i}
            end if

        else                                    /* If fault f_j is activated by T_i */
            if (f_j ∈ F_NF) then                /* If fault f_j is a non-feedback fault */
                                                /* eliminate it */
                eliminate f_j
            else                                /* If f_j is a feedback fault */
                if (f_j is sensitized) then     /* If f_j is sensitized, place it in F_S */
                    f_j ∈ F_{S_i}
                else                            /* If f_j is not sensitized, eliminate it */
                    eliminate f_j
                end if
            end if
        end if
    end if
  end for
end for
end procedure                                   /* End CLASSIFY */


procedure UNDETECTED()                          /* Identifies all faults which are */
                                                /* undetectable and places them in F_U */

F_{U(0)} = F_P
for i = 1 to t                                  /* Analysis is over all t test vectors */
  for j = 1 to TOTP                             /* Analyse all faults in F_P */
```

```
        actflag = ACTIVATION(f_j)          /* Determine if fault f_j is activated by */
                                           /* the current test vector */
        if (actflag = true) then           /* Fault f_j is activated by T_i */
            eliminate f_j from F_U         /* Fault f_j remains in list F_P */
        else                               /* Fault f_j is not activated by T_i */
            f_j ∈ F_U(i)                   /* Fault f_j is placed in list F_U and */
                                           /* removed from list F_P */
        end if
    end for
end for
end procedure                              /* End UNDETECTED */


procedure AP_UNIQUE()                      /* Determines if any fault */
                                           /* in F_P and F_S is solely responsible */
                                           /* for the I_DDQ fault observed in the */
                                           /* CUT during the application of a */
                                           /* particular test vector */


for i = 1 to t
    if (i ∈ AP_CUT) then                   /* Analysis is done for all test vectors */
                                           /* which produce an I_DDQ fault */
        q = 0
        for j = 1 to TOTI
        actflag = ACTIVATION(f_j)          /* Determine if fault f_j is activated by */
                                           /* the current test vector */
            if (f_j ∈ {F_P ∪ F_S}) then    /* Analyse f_j only if it is in either */
                                           /* F_P or F_S */
                if (actflag = true) then   /* If f_j is activated by the current test */
                                           /* vector, save the fault index (j) and */
                                           /* increment counter q */
                    num = j
                    q = q + 1
                end if
                if (q ≥ 2) then end for    /* If more than one fault is activated by */
                                           /* the current test vector, stop the */
                                           /* analysis of the current test vector */
                                           /* and move on to the next one */
                end if
            end if
        end for
        if (q = 1) then                    /* If only one fault is activated by T_i, */
                                           /* move that fault to F_L */
            f_num ∈ F_L
        end if
    end if
end for
```

```
end procedure                          /* End AP_UNIQUE */

procedure NTUPLE(x)                    /* Multiple fault analysis to create fault */
                                       /* list F_N(x). x is the degree of */
                                       /* n-tuple fault to be analysed. */
                                       /* (i.e. for double faults x=2, */
                                       /* for triple faults x=3, etc.) */

INDEX = a set of integers              /* Initial classifications */
F_SP = (F_S \ F_SE) ∪ F_P
TOTSP = TOT(S \ SE) + TOTP is the number of faults in F_SP
F_N0(x) = {f_n1, f_n2, ..., f_nTOTN} where TOTN = C_x^TOTSP
f_nj = {f_n1j, ..., f_nxj}

for i = 1 to t                         /* Analysis is over all t test vectors */
   if (i ∈ AP_CUT) then                /* If an I_DDQ fault is detected in the */
                                       /* CUT by T_i */
      for j = 1 to TOTSP               /* Analyse all faults in F_SP */
      actflag = ACTIVATION(f_j)        /* Determine if f_j is activated by T_i */
         if (actflag = false) then     /* If-f_j is not activated, store the fault */
                                       /* index (j) in INDEX */
            j ∈ INDEX
         end if
      end for
      for s = 1 to TOTN                /* Analyse all possible multiple faults */
         if (n_s1, ..., n_sx ∈ INDEX) then  /* If all single faults in multiple fault f_ns */
                                       /* are not activated by the current test */
                                       /* vector, eliminate the multiple fault */
            eliminate f_ns
         end if
      end for
   end if
end for
end procedure                          /* End NTUPLE */
```

3.5.2 Multiple Fault Diagnosis Example

Since the CLASSIFY procedure is similar to the single fault algorithm, a detailed example of it will not be presented here. It is also assumed for this example that the UNDETECTED and AP_UNIQUE procedures have already been completed. The example in this subsection illustrates how the NTUPLE procedure uses the information produced by the other procedures to identify the possible n-tuple combinations

which can produce the I_{DDQ} test results observed from the CUT.

Table 3.4 illustrates the fault sets that are created from the first three procedures of the multiple fault algorithm. For each fault f_j, AP_{SIM_j} is given to help illustrate the NTUPLE procedure.

$AP_{CUT} = \{2, 4, 6, 7\}$							
F_L		F_P		F_S		F_U	
Fault	AP_{SIM_j}	Fault	AP_{SIM_j}	Fault	AP_{SIM_j}	Fault	AP_{SIM_j}
$f1$	{2, 4, 6, 7}	$f2$	{2, 4}	$f3$	{1, 4}	$f5$	{∅}
$f19$	{2, 4, 6, 7}	$f6$	{6, 7}	$f4$	{4, 5, 6, 7}	$f7$	{∅}
		$f10$	{7}	$f9$	{2, 4, 6, 7, 10}	$f8$	{∅}
		$f11$	{2, 6}	$f12$	{2, 3, 4, 6, 7, 9, 10}		
		$f14$	{4, 6, 7}	$f13$	{3, 4, 6}		
		$f15$	{4}	$f17$	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}		
		$f16$	{6, 7}	$f18$	{6, 7, 8, 9, 10}		
				$f20$	{2, 4, 9, 10}		

Table 3.4: Input to NTUPLE Procedure for Multiple Fault Example

Since faults $f1$ and $f19$ have $AP_{SIM_j} = AP_{CUT}$, they are not analysed by procedure NTUPLE. This is because they alone can be responsible for the faulty I_{DDQ} tests produced from the CUT. Therefore, it is worthless to combine them with other faults, as they will "mask out" the effects of the other faults. This masking effect also occurs for faults $f9$, $f12$, and $f17$ from set F_S (these three faults are in F_{SE}). Consequently, they are not analysed by procedure NTUPLE. The faults in F_U, ($f5$, $f7$, and $f8$,) are never activated by the test vectors used during the I_{DDQ} tests. They are also excluded from procedure NTUPLE. The faults which are analysed are the faults from F_P ($f2$, $f6$, $f10$, $f11$, $f14$, $f15$, $f16$) and the remaining faults from F_S ($f3$, $f4$, $f13$, $f18$, $f20$). These 12 faults make up fault set F_{SP} defined in procedure NTUPLE.

For this example, a double fault assumption is used. Each possible double fault is analysed (a total of $C_2^{12} = 12!/(2! \times 10!) = 66$) for every test vector T_i, where $i \in AP_{CUT}$ (2, 4, 6, 7), until either all the information from each test vector has been analysed, or the double fault is eliminated. The analysis proceeds as follows.

$i = 2$:

Since faults $f2$, $f11$ and $f20$ are the only faults which have $2 \in AP_{SIM_j}$, any of the

double fault combinations that do not contain one of $f2$, $f11$, or $f20$ are eliminated. The other double faults remain in $F_N(2)$. Therefore, the following double faults are left in $F_N(2)$:

$F_N(2)$ = {$f2$-$f3$, $f2$-$f4$, $f2$-$f6$, $f2$-$f10$, $f2$-$f11$, $f2$-$f13$, $f2$-$f14$, $f2$-$f15$, $f2$-$f16$, $f2$-$f18$, $f2$-$f20$, $f3$-$f11$, $f3$-$f20$, $f4$-$f11$, $f4$-$f20$, $f6$-$f11$, $f6$-$f20$, $f10$-$f11$, $f10$-$f20$, $f11$-$f13$, $f11$-$f14$, $f11$-$f15$, $f11$-$f16$, $f11$-$f18$, $f11$-$f20$, $f13$-$f20$, $f14$-$f20$, $f15$-$f20$, $f16$-$f20$, $f18$-$f20$}

$i = 4$:

The single faults for which $4 \in AP_{SIM_j}$ are $f2$, $f14$, $f15$, $f3$, $f4$, $f13$, and $f20$. Therefore, only double faults in $F_N(2)$ containing at least one of these faults are kept. After the information from test vector 4 has been analysed, the following faults remain in $F_N(2)$:

$F_N(2)$ = {$f2$-$f3$, $f2$-$f4$, $f2$-$f6$, $f2$-$f10$, $f2$-$f11$, $f2$-$f13$, $f2$-$f14$, $f2$-$f15$, $f2$-$f16$, $f2$-$f18$, $f2$-$f20$, $f3$-$f11$, $f3$-$f20$, $f4$-$f11$, $f4$-$f20$, $f6$-$f20$, $f10$-$f20$, $f11$-$f13$, $f11$-$f14$, $f11$-$f15$, $f11$-$f20$, $f13$-$f20$, $f14$-$f20$, $f15$-$f20$, $f16$-$f20$, $f18$-$f20$}

$i = 6$:

The single faults for which $6 \in AP_{SIM_j}$ are $f6$, $f11$, $f14$, $f16$, $f4$, $f13$, and $f18$. Once again, only double faults containing at least one of these faults are kept. The faults remaining in $F_N(2)$ are:

$F_N(2)$ = {$f2$-$f4$, $f2$-$f6$, $f2$-$f11$, $f2$-$f13$, $f2$-$f14$, $f2$-$f16$, $f2$-$f18$, $f3$-$f11$, $f4$-$f11$, $f4$-$f20$, $f6$-$f20$, $f11$-$f13$, $f11$-$f14$, $f11$-$f15$, $f11$-$f20$, $f13$-$f20$, $f14$-$f20$, $f16$-$f20$, $f18$-$f20$}

$i = 7$:

Test vector 7 is the last test vector that produces an I_{DDQ} fault in the CUT. The faults for which $7 \in AP_{SIM_j}$ are $f6$, $f10$, $f14$, $f16$, $f4$, and $f18$. Once the double faults which do not contain at least one of these single faults are eliminated, $F_N(2)$ contains the following double faults:

$F_N(2)$ = {$f2$-$f4$, $f2$-$f6$, $f2$-$f14$, $f2$-$f16$, $f2$-$f18$, $f4$-$f11$, $f4$-$f20$, $f6$-$f20$, $f11$-$f14$, $f14$-$f20$, $f16$-$f20$, $f18$-$f20$}

This completes the operations performed by procedure NTUPLE. The remaining

faults in $F_N(2)$ are the double faults which may be responsible for producing all of the I_{DDQ} failures. Table 3.5 presents the final results of the multiple fault algorithm. Simulations to test the multiple fault algorithm are presented in chapter 4.

$F_N(2)$	
F_{NP}	F_{NS}
$f2$-$f6$	$f2$-$f4$
$f2$-$f14$	$f2$-$f18$
$f2$-$f16$	$f4$-$f11$
$f11$-$f14$	$f4$-$f20$
	$f6$-$f20$
	$f14$-$f20$
	$f16$-$f20$
	$f18$-$f20$

Table 3.5: Results of NTUPLE Procedure for Multiple Fault Example

3.6 Complexity Analysis of Algorithms

The complexity of an algorithm is measured by determining the total number of *basic operations* $t(I)$ that the algorithm performs on an input I from a set of input data D_n [8]. A basic operation is chosen so that the number of basic operations is proportional to the total number of operations performed by the algorithm.

When analysing the number of basic operations, the worst case scenario is assumed. This allows for a conservative estimate for the complexity. Once the *worst case complexity* $W(n)$ has been determined, an expression for the *order* or growth rate of each algorithm can be determined. In [8], the worst case complexity is computed as

$$W(n) = max\{t(I)|I \in D_n\}. \tag{3.1}$$

By ignoring constants in the expression for $W(n)$, an expression in "big O" notation can be found.

3.6.1 Single Fault Algorithm

The basic operation performed by the single fault algorithm is a decision or action based on the simulation information and test results from the CUT. The two sources

of input data which can vary between diagnosis runs on the same CUT are the number of input test vectors (t), and the number of faults in the initial fault list (n).

In the worst case scenario, every fault in the initial fault list is a feedback fault which is sensitized by every test vector in the input test set. In addition, none of the test vectors in the test set produce an I_{DDQ} fault in the CUT ($AP_{CUT} = \emptyset$). If the algorithm is invoked under such circumstances, no faults are eliminated during a diagnosis session, and all of the loops in the algorithm are traversed the maximum number of times.

The following operations are performed. AP_{CUT} is analysed to see if the current test vector produces an I_{DDQ} fault in the CUT. This operation is performed t times (once for each test vector). Each fault that has not been eliminated must be checked to see whether it is activated by the current test vector. When no faults are eliminated, this operation is performed $n \times t$ times. Finally, each feedback fault must be checked to determine whether it is sensitized or not. This operation is also performed $n \times t$ times.

Given the basic operations above, the expression for the worst case complexity is $W(n) = t + 2tn$. It can be seen that the complexity of the algorithm varies linearly with both the number of test vectors and the number of faults in F_I. Since the vast majority of faults are eliminated within the first 10 test vectors during an actual diagnosis, the n is much less than in the worst case scenario we assumed. For a specific diagnosis run, t is constant. Therefore, the complexity of the algorithm is $O(n)$.

3.6.2 Multiple Fault Algorithm

Since the multiple fault algorithm consists of four procedures, the complexity analysis is performed separately in stages.

procedure CLASSIFY

This procedure is very similar to the single fault algorithm. Therefore, $W(n) = t + 2tn$, and the complexity is $O(n)$.

procedure UNDETECTED

In this procedure, faults from set F_P are examined to identify faults that are never activated by the current test set. The basic operation performed is the analysis of

AP_{SIM_j} for each fault to determine whether it is activated by the current test vector. This operation is performed $n \times t$ times. This makes $W(n) = tn$, and the complexity is $O(n)$.

procedure AP_UNIQUE

If F_L is empty, the faults in F_P and F_S are analysed to determine if any one fault from these sets is solely responsible for the faulty I_{DDQ} results observed in the CUT for a particular input test vector. The basic operation for this procedure is analysing every fault in F_P and F_S to see whether they are activated or not. The maximum number of times this operation is performed is $n \times t$. Therefore, the worst case complexity is $W(n) = nt$, and the complexity for the step is $O(n)$.

procedure NTUPLE

In procedure NTUPLE, logic simulation information is used to determine which multiple faults formed from fault lists F_P and F_S may be responsible for the faulty I_{DDQ} results observed during testing. The basic operations for this procedure involve the analysis of AP_{SIM_j} for each single fault. The two operations performed are: checking which single faults are activated by each test vector, and determining which possible multiple fault combinations can be eliminated. The first operation is performed up to $n \times t$ times, while the second is performed up to C_x^n times, where x is the degree of multiple fault considered. Taking all basic operations into account, the expression for the worst case complexity is $W(n) = nt + n!/(x!(n-x)!)$. Ignoring constants, the expression for the complexity of this procedure is approximately $O(n^x)$ for $n >> x$.

Since procedure NTUPLE provides the maximum complexity of all the procedures, the complexity of the multiple fault algorithm is $O(n^x)$. Again, in an actual diagnosis run, most of the faults in the initial fault list are eliminated within 10 test vectors (i.e. $n << TOTI$), and most CUTs which are tested will only contain a single fault, double fault, or triple fault (i.e. $x \leq 3$).

3.7 Factors Affecting Algorithm Performance

Many factors can affect the performance achieved by the diagnosis algorithms. In this section we first discuss design considerations which help improve the I_{DDQ} testa-

bility of the CUT, improve the diagnostic resolution, and minimize the possibility of misdiagnosis. We then outline other design considerations which will not affect the testability of the CUT, but still improve the resolution and quality of diagnosis. Finally, implementation aspects which affect the amount of memory and time required for each diagnosis session such as test vector re-ordering, and the assumptions make to deal with feedback faults are discussed.

3.7.1 Design Recommendations

Design rules which either eliminate undetectable faults or increase the I_{DDQ} testability of the CUT will help to increase the resolution provided by the algorithms and minimize the time required to execute a diagnosis session. We provide three design recommendations that will help maximize the performance achieved by the diagnosis algorithms.

The first recommendation is to design the CUT using as few inverters as possible. Reducing the number of inverters in the CUT leads to a better diagnostic resolution. The reason for this is illustrated below using figure 3.3.

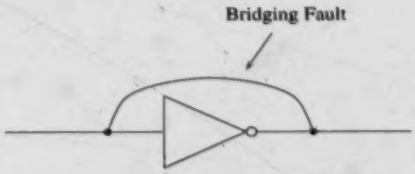


Figure 3.3: Feedback Faults on Inverters

In figure 3.3, a feedback fault exists between the input and the output of an inverter. Recall from chapter 2 that to activate a bridging fault, the two nodes at either end of the fault must be set to opposite logic values. Therefore, the feedback fault in figure 3.3 is activated regardless of which input vector is applied to the circuit. It is also obvious that the fault is sensitized by every test vector. Recall that feedback faults which are activated by test vectors that do not produce faulty I_{DDQ} levels are eliminated only when the fault is not sensitized. Consequently, bridges between the input and output nodes of an inverter can never be eliminated by the diagnosis algorithms.

During a diagnosis run, every bridge between an inverter input and output will be placed into fault set F_S. If the CUT has a large number of inverters, a large number of faults will be left after a diagnosis session and the resolution will be poor. If circuits are designed using a minimum number of inverters, this problem will be alleviated, and the resolution will be improved.

The second recommendation to improve the performance of the algorithms is to eliminate redundant nodes. If a fault exists between two nodes in the CUT which share the same logic value for all test vectors in the test set (logically equivalent nodes), the fault can never be activated. By avoiding redundancies, the number of undetectable bridges is reduced, and a higher quality of diagnosis is obtained. An approach to remove circuit redundancies is presented in [42].

The final recommendation is to design all CUT's using fully complementary standard cell CMOS gates. If standard cells are used in design, the I_{DDQ} testability of the CUT is assured. Using non-standard CMOS gates or logic blocks in a design can introduce situations which are detrimental to I_{DDQ} tests. For example, circuits using pre-charge logic can experience charge sharing between nodes, which may lead to faulty I_{DDQ} levels being measured in a fault-free circuit.

A previous work [21] has defined a minimal set of design rules which ensure that circuits are highly I_{DDQ} testable. By using standard cell libraries, the following rules from [21] are satisfied.

1. The gate and drain (or source) nodes of a transistor are not in the same transistor group. A transistor group is the set of transistors which are connected together to form a conducting channel [21].

2. During steady-state operation, there is no conducting path from Vdd to GND.

3. During steady-state operation, each output of a transistor group must be connected to Vdd or GND through a path of conducting transistors.

3.7.2 Efficiency Improvement Using Test Vector Re-ordering

The CPU time and memory required to perform a diagnosis run contribute to the feasibility and quality of a diagnosis method. For the multiple fault algorithm, both

can be greatly affected by the order in which the test vectors applied to the CUT are analysed during a diagnosis session.

If the majority of faults under consideration are eliminated shortly after the algorithm is invoked, the time and memory requirements decrease significantly. In the multiple fault algorithm, faults can only be eliminated when a particular test vector does not produce a faulty level of I_{DDQ} in the CUT. If all of the test vectors near the beginning of the test set produce faulty levels of I_{DDQ}, the initial fault set will not be reduced until each of these vectors has been analysed. As a result, the amount of time and computer memory required to process the diagnostic information is maximized as information about every fault must be computed and stored.

Since it is known which test vectors produce a failing level of I_{DDQ} before the multiple fault algorithm is invoked, the above problems can be avoided. If all test vectors which do not cause a faulty level of I_{DDQ} are processed before the other test vectors in the test set, every fault which does not end up in one of the diagnosed fault sets (F_L, F_S, or F_P, or F_U) will be eliminated directly. Since AP_{SIM_j} is not stored for eliminated faults, the amount of required computer memory will be minimized. The trade-off for the savings of memory and CPU time is the amount of time required to reposition the test vectors and their corresponding I_{DDQ} test results from the CUT before they are passed to the algorithm. In the vast majority of cases, the amount of time and memory saved far outweighs the extra effort required to re-order the test vector set.

3.7.3 Effects of Feedback Fault Assumptions

The new diagnosis algorithms assume that all current testing is performed during the steady-state using simple current sensors or off-line testers. Moreover, both of the diagnosis algorithms assume that activated, sensitized feedback faults do not necessarily produce faulty levels of I_{DDQ}. Due to this conservative assumption, the number of faults appearing in set F_S can be relatively large. However, depending on the testing equipment available, previous studies have suggested that feedback faults may not have to be treated in such a pessimistic manner. Feedback faults which do not produce easily measurable values of I_{DDQ} due to oscillations and sensitized feedback paths can usually be detected by monitoring or recording the average power

supply current [24, 35]. If all activated feedback faults are detectable using current monitoring methods, feedback faults can be treated by the algorithms exactly the same way as non-feedback faults. Therefore, the F_S fault set will no longer be required for either the single fault or multiple fault algorithms. This would lead to a faster diagnosis as sensitization checks and feedback path checks will no longer be necessary during diagnosis, and a higher diagnostic resolution is obtained since more feedback faults will be eliminated. On the other hand, this would require either a significant increase in hardware due to the complex current sensors required to measure the average current, or a complex off-line tester to perform the current measurements.

3.7.4 Effects of Using Different Test Sets

Using different sets of test vectors influences both the quality of testing and the quality of diagnosis. A good test set should be able to activate each fault in the circuit at least once, and should also contain enough test vectors so that most faults will produce unique values for AP_{SIM_f}. However, test sets should not be too long as the time required to perform a test should be as small as possible. Consequently, the length of a test set is a trade-off between test application time and diagnostic resolution. Three types of test vector sets which are commonly used in testing schemes are discussed below.

Exhaustive Test Set

In an exhaustive test set, all possible test vectors are used to test and diagnose the CUT (i.e. for a CUT with n inputs, 2^n test vectors are used). Exhaustive sets provide the best diagnostic results as they produce the maximum amount of information available. The obvious problem with using an exhaustive test set is that for large circuits, exhaustive tests are impossible due to the amount of time required. Since most practical circuits contain a large number of inputs, exhaustive testing is not usually used.

Pseudorandom Test Set

Pseudorandom tests are commonly used in testing and diagnosis schemes. Pseudorandom test sets are easily generated, and provide adequate fault coverage if long

enough test sequences are used. For pseudorandom test sets, the following factors influence the diagnostic resolution provided by the diagnosis algorithms.

1. The number of vectors in the test set.
2. The pseudorandom pattern generator (PRPG) used to generate the pseudorandom sequences (i.e. linear feedback shift register (LFSR), linear cellular automata register (LCAR) etc. [10]).
3. The feedback polynomial used to define the PRPG [10].
4. The initial state of the PRPG.

Each of these factors helps to determine the number of times that each bridging fault in the circuit is activated during a test session. When the number of test vectors used for diagnosis is increased, better resolution is expected as AP_{SIM_j} for each fault in the CUT will become more unique. Using different PRPGs or different feedback polynomials for the same type of PRPG may alter the randomness of the input test set, and consequently make each AP_{SIM_j} more unique. Similarly, using a different initial state for the PRPG will produce a different input test set, which may improve the resolution achieved by the algorithm.

I_{DDQ} Test Set

Special test sets can be generated specifically for I_{DDQ} testing purposes. I_{DDQ} test sets are generally very small compared to the voltage-based test sets which are used to test the same circuits. For example, the I_{DDQ} test sets generated using the method in [26] can be approximately 1% of the size of stuck-at fault test sets.

Although a high percentage of bridging fault coverage can be achieved from I_{DDQ} test sets, they may not be good for use with the diagnosis algorithms. Since the test sets are so short, many bridging faults may share the same AP_{SIM_j}. In this case, the diagnostic resolution will be relatively poor as many faults will appear in fault set F_L for each different AP_{CUT}.

3.8 Diagnosis Beyond Algorithms

As stated in chapter 2, feedback bridging faults may or may not cause I_{DDQ} faults to occur even when the fault's component nodes are set to opposite logic values. Both the single fault algorithm and the multiple fault algorithm assume that if a feedback fault is sensitized, the effect that the fault has on a circuit cannot be determined using logic simulation. Consequently, both algorithms place many sensitized feedback faults into fault set F_S instead of eliminating them.

To determine the effects of sensitized faults, circuit-level fault simulation can be performed. Using circuit-level simulation on the fault sets produced by each algorithm will also identify the faults which do not create identical logic results to those observed in the CUT. These faults can then be eliminated, and the diagnostic resolution can be increased.

Once the number of diagnosed faults has been minimized using fault simulation, electron beam testing can be used to search for the physical circuit defects responsible for all faulty behaviour. The following subsections suggest procedures for analysing the outputs of both the single and multiple fault algorithms.

3.8.1 Single Fault Algorithm Output

The output of the single fault algorithm consists of two fault lists: F_L and F_S. Recall that F_L contains all faults in the CUT for which AP_{SIM_i} matches AP_{CUT} exactly, while F_S contains feedback faults where $AP_{SIM_i} \neq AP_{CUT}$ (see page 34). Therefore, a single fault in the CUT is far more likely to be in F_L than in F_S. Consequently, we suggest the following methodology for subsequent diagnosis.

1. Perform fault simulation on the CUT using only those faults from list F_L.
2. If no faults from F_L produce identical results to those of the CUT, perform the above steps using the faults from set F_S.

Analysing set F_L before set F_S will decrease the total fault simulation time for the following reasons. The fault in the CUT is most likely to be in F_L, so it will be located faster. Secondly, simulating feedback faults with a circuit-level simulator is

difficult and time consuming due to the feedback loops that are set up in the circuit. Therefore, if a fault can be located before set F_S is analysed, much time will be saved.

Once fault simulation is complete, electron beam testing can be used to investigate the circuit nodes on all of the diagnosed faults.

3.8.2 Multiple Fault Algorithm Output

If diagnosis using the single fault algorithm does not provide good results (i.e. no faults appear in list F_L), the diagnosis can be repeated using the multiple fault algorithm. The multiple fault algorithm produces the following fault sets: F_L, F_S, F_P, F_U, F_{NP}, and F_{NS}. The following is a suggested procedure for subsequent diagnosis.

1. Fault list F_L is analysed the same way as is done for the single fault algorithm. If the single fault algorithm has already been used on the CUT, F_L will contain exactly the same information, so it does not have to be analysed again.

2. Perform fault simulation using the multiple faults in set F_{NP}. Each simulation is performed by injecting all of the single faults into a copy of the CUT which combine together to form a multiple fault.

3. Simulations can either be stopped if a multiple fault is found which produces output logic values identical to those from the CUT, or they can be performed on all faults regardless of whether a match is found.

4. If no multiple faults from set F_{NP} give the same outputs as those from the CUT, simulations should proceed using the multiple faults from fault set F_{NS}.

This method will help minimize the amount of time required to perform the simulations, as the faults in set F_{NP} contain the faults which are most likely to exist in the CUT. Faults from set F_P do not have to be simulated individually as they cannot be solely responsible for all of the I_{DDQ} faults in the CUT. Similarly, single faults from set F_S do not have to be simulated as either they cannot be solely responsible for the faulty behaviour in the CUT, or they have already been covered during the analysis of the single fault algorithm output sets. Obviously, faults from set F_U do not have to be analysed, as they are never activated during a test session. Once the information

from fault simulation has been collected, the nature of the physical circuit defects can be analysed with an electron beam tester.

Chapter 4

Simulations

To evaluate the effectiveness of the new diagnosis algorithms, computer simulations were performed on some standard benchmarks circuits. The simulations help to illustrate the performance of the algorithms and also show that it is feasible to apply the algorithms on large VLSI circuits.

An overview of the simulation environment is given, including discussions about the circuits used for the simulations and the computer resources available. The simulation goals are defined and the simulation results are presented. Using the simulation results, the performance of the algorithms is analysed.

4.1 Simulation Environment

Figure 4.1 shows an overview of our simulation environment. The center of the environment is a logic simulator named BRIDGE [32]. It was developed to implement the diagnosis algorithms in chapter 3. BRIDGE requires three sources of input to perform a diagnosis session: a circuit netlist, a set of input test vectors, and a corresponding set of I_{DDQ} test results (AP_{CUT}). The main outputs produced by BRIDGE are the fault sets described in chapter 3 (F_L and F_S for the single fault algorithm, and F_L, F_P, F_S, F_U, and $F_N(x)$ for the multiple fault algorithm). The implementation issues of BRIDGE are treated separately and are discussed later in this chapter. The user's manual of BRIDGE can be found in appendix B.

The circuits used for the simulations are from the standard Berkeley and ISCAS85 benchmarks [12, 41]. These benchmarks are sets of digital circuits used by industry in the late 1970's and mid 1980's, respectively. For the past two decades, they have been

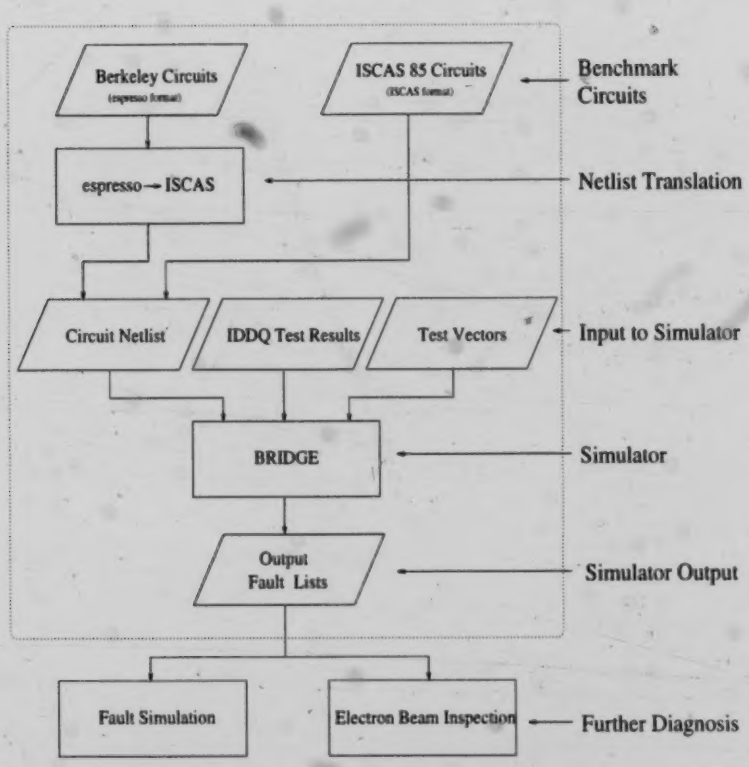


Figure 4.1: Overview of Simulation Environment

used as the standard benchmark circuits to evaluate logic minimization algorithms, design and test strategies, and simulation algorithms.

The Berkeley benchmarks are significantly smaller in size and implement simpler logic functions than the ISCAS85 benchmarks. This allows for exhaustive tests and simulations. On the other hand, ISCAS85 benchmarks are sophisticated VLSI circuits for which exhaustive simulations are impossible. Consequently, BIST and DFT techniques are normally used to enhance the testability of the ISCAS85 circuits. Using both sets of benchmarks for our experiments provides both thorough and realistic evaluations of our diagnosis algorithms.

Berkeley benchmarks are given in *espresso* format (i.e. truth table format), which can be directly mapped into PLA (programmable logic array) implementations. The ISCAS85 benchmarks are expressed in a netlist format of a multiple-level gate implementation. Multiple-level gate implementations allow for the use of standard cell

libraries in designs, require less silicon area than that of PLA implementations, and are commonly used in today's computer-aided VLSI designs.

The BRIDGE simulator accepts circuits that are expressed in the ISCAS85 netlist format. To convert the Berkeley benchmarks to the ISCAS85 netlist representation they are first translated to the OASIS netlist format using *decaf* [29]. This converts each circuit from a truth-table format to a logically equivalent multiple-level gate implementation. Second, the benchmarks are translated from the OASIS netlist format to the ISCAS85 netlist format. Table 4.1 lists each of the circuits used for the simulations and shows the number of feedback faults and non-feedback faults that can exist in each circuit.

	Benchmark	Inputs	Outputs	Logic Gates	Feedback Faults	Non-feedback Faults	Total Faults
B	f2	4	4	24	162	216	378
E	rd53	5	3	41	321	714	1035
R	rd73	7	3	126	1662	7116	8778
K	sao2	10	4	146	1976	10114	12090
E	bw	5	28	155	1421	11299	12720
L	rd84	8	4	219	3990	21661	25651
E	9sym	9	1	231	3724	24717	28441
Y							
	c432	36	7	160	9978	9132	19110
I	c499	41	32	202	12722	16681	29403
S	c880	60	26	383	16004	81899	97903
C	c1355	41	32	546	81826	90165	171991
A	c1908	33	25	880	108912	307416	416328
S	c2670	233	140	1193	52558	963467	1016025
8	c3540	50	22	1669	235584	1241037	1476621
5	c5315	178	123	2307	109084	2977286	3086370
	c7552	207	108	3512	221275	6692346	6913621

Table 4.1: Characteristics of Benchmark Circuits

4.2 Simulation Goals and Assumptions

The goal of the simulations is to demonstrate the feasibility of the algorithms given in chapter 3. The simulations investigate the diagnostic resolution produced for each benchmark circuit in table 4.1 when either the single fault assumption is employed, or when a double fault assumption is employed. The CPU time required to perform each diagnosis run is also recorded. For all simulations it is assumed that no layout

information is available for each CUT. Therefore, every two-node, gate-level bridge is considered during each diagnosis run.

4.3 Simulation Results

This section presents the results of the simulations for both the single fault algorithm and the multiple fault algorithm. Tables containing the complete results from the simulations can be found in appendix C and appendix D.

4.3.1 Sample I_{DDQ} Test Result Generation

Both the single fault algorithm and the multiple fault algorithm require I_{DDQ} test results (AP_{CUT}) to perform a diagnosis. In an actual diagnosis environment, AP_{CUT} is generated from the faulty circuit(s) being diagnosed. Since no faulty circuits are available for our experiments, AP_{CUT} is obtained from other sources. We generate different values for AP_{CUT} by simulating single bridging faults in each of the benchmark circuits, and recording the expected I_{DDQ} test results, as described below.

Single Fault AP_{CUT}

For each Berkeley and ISCAS benchmark, 30 faulty circuits were simulated. For each faulty circuit, a single two-node bridge was chosen at random. Using logic simulation, the expected AP_{CUT} was generated by recording the test vectors in the test set that activate the bridge (i.e. the two component nodes of the bridge are set to opposite logic values). For ease of presentation, logically simulating a bridging fault in a CUT will be referred to as *injecting* a fault into the CUT for the remainder of this thesis unless otherwise specified.

Each test result is initially stored as a binary sequence where a 1 indicates that the circuit being simulated failed the I_{DDQ} test for the corresponding input vector and a 0 indicates that the circuit passed the I_{DDQ} test for that vector. AP_{CUT} is formed from each sequence by saving the indices of each sequence entry that is a 1.

Multiple Fault AP_{CUT}

Similarly, sample I_{DDQ} test results are also required for circuits containing 2 or more bridges. For each benchmark circuit, 10 double fault I_{DDQ} test results were generated. Double fault I_{DDQ} test results were formed by bit-wise ORing two single fault patterns together. The resulting pattern is used as the one from a circuit containing both of the single faults in question. The corresponding AP_{CUT} is obtained by saving the indices of the sequence entries that are 1's. To generate multiple faults of higher degrees (i.e. triple faults, quadruple faults), the same methodology (bit-wise ORing) can be used.

Test Length

The total time required to execute a diagnosis run depends upon the number of test vectors used during the I_{DDQ} test session. To minimize the testing time, the total number of vectors should be as small as possible. However, as was mentioned in chapter 3, the test set should be long enough to provide an acceptable diagnostic resolution (i.e. a high percentage of the faults in the initial fault list are eliminated during diagnosis).

To illustrate the test length required to eliminate the majority of faults in the initial fault list, two simulations were performed using the multiple fault algorithm as this algorithm eliminates fewer faults during its execution (i.e. the worst case scenario). Each simulation performs a diagnosis run on one of the ISCAS85 benchmarks (c432 and c5315) while the number of faults eliminated after each test vector was analysed was recorded. The percentage of the total faults in the CUT that were eliminated was computed for test lengths of up to 90 test vectors. The graphs in figures 4.2 and 4.3 illustrate the results of the test length simulations.

Figures 4.2 and 4.3 show that the majority of the diagnosis information can be obtained using relatively short test lengths since well over 90% of the faults are eliminated within 90 test vectors. Although the diagnostic resolution can be improved by using more test vectors for a diagnostic session, the number of vectors required for even a modest improvement in resolution can greatly increase the time required for diagnosis.

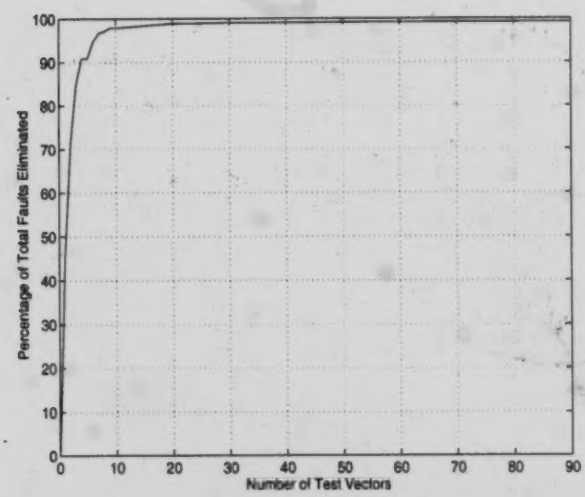


Figure 4.2: Faults Eliminated After 90 Test Vectors for c432

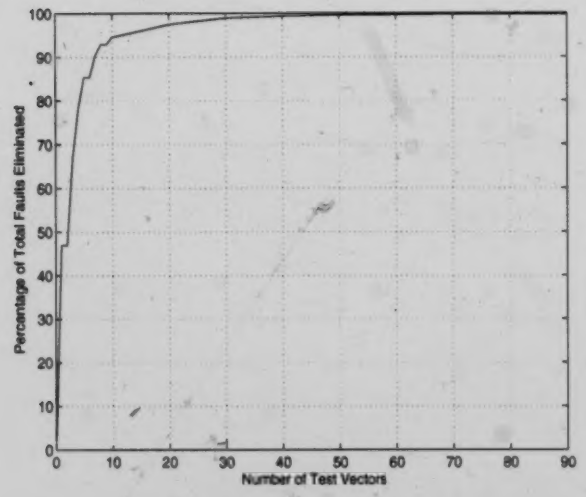


Figure 4.3: Faults Eliminated After 90 Test Vectors for c5315

For the simulations in this chapter, the test length used for each benchmark must be chosen. For the Berkeley benchmarks, the number of inputs that each circuit has is small enough that exhaustive sets of test vectors are used to generate the I_{DDQ} test results. The number of vectors used for each Berkeley circuit is listed in table 4.2.

Benchmark	Inputs	Number of Test Vectors
f2	4	16
rd53	5	32
rd73	7	128
sao2	10	1024
bw	5	32
rd84	8	256
9sym	9	512

Table 4.2: Number of Test Vectors used for Berkeley Benchmark Simulations

For the ISCAS85 benchmarks, each circuit has too many inputs for exhaustive test sets to be employed. For these circuits, a test length of 1000 pseudorandom vectors was chosen for use in the simulations. As suggested by the results in figure 4.2 and figure 4.3, 1000 test vectors should be a sufficient test length to produce a high diagnostic resolution.

Each set of vectors is generated using a minimum cost LFSR defined by a primitive polynomial [10]. The length of the LFSR used for each benchmark circuit is equal to the number of primary inputs in the circuit. Table 4.3 shows the number of inputs for each ISCAS85 circuit, as well as the polynomial which describes the LFSR used to generate the input vectors. These polynomials are taken from the list of primitive polynomials given in [10].

4.3.2 Simulations Using the Single Fault Algorithm

Three different sets of simulations were performed to evaluate the single fault algorithm. A set of simulations using the single fault I_{DDQ} test results was performed to examine the effectiveness of the single fault algorithm. A second set of simulations using the same test set lengths as used in [14] was performed in order to compare our algorithm with the single fault algorithm in [14]. The third set of simulations was performed using double fault I_{DDQ} test results to illustrate the possibility of

Benchmark	Inputs	LFSR Polynomial	Number of Test Vectors
c432	36	$x^{36}+x^{11}+1$	1000
c499	41	$x^{41}+x^{3}+1$	1000
c880	6	$x^{60}+x+1$	1000
c1355	41	$x^{41}+x^{3}+1$	1000
c1908	33	$x^{33}+x^{13}+1$	1000
c2670	233	$x^{233}+x^{74}+1$	1000
c3540	50	$x^{50}+x^{27}+x^{26}+x+1$	1000
c5315	178	$x^{178}+x^{87}+1$	1000
c7552	207	$x^{207}+x^{43}+1$	1000

Table 4.3: LFSR Polynomials Used for ISCAS85 Benchmark Simulations

misdiagnosis when using the single fault algorithm and demonstrate the necessity of having the multiple fault algorithm. Tables containing the results of all simulations performed using the single fault algorithm are found in appendix C. All simulations using the single fault algorithm were performed on a SPARC 5 workstation having 64 MBytes of RAM.

Single Fault Simulations

The diagnostic resolution provided by the algorithms was examined by analysing the output fault sets (F_L and F_S). If the sets contain a small number of faults, the diagnostic resolution is high. The higher the diagnostic resolution is, the more valuable the diagnostic information becomes for finding actual defects in the CUT. For each of the simulations, the CPU time required to perform the diagnosis was also recorded to help illustrate the feasibility of the algorithms in diagnosing large circuits.

The results of the first set of simulations are summarized in table 4.4. The numbers in columns F_L and F_S are the averages for the 30 simulation runs performed on each benchmark. For example, for benchmark f2, the average number of located faults (faults whose simulated I_{DDQ} test results match exactly with those given by AP_{CUT}) is 1.667. Similarly, the average number of sensitized feedback faults is 9.000. Finally, the average time required for a diagnosis run for benchmark f2 is 0.070 seconds.

The values in table 4.4 show that the single fault algorithm is effective at reducing the large, exhaustive set of bridging faults to a much smaller set of diagnosed faults for both Berkeley and ISCAS85 benchmark circuits. For the Berkeley benchmarks, the totals for F_L indicate that, in most cases, fewer than two faults produce identical

Circuit	Test Length	F_L	F_S	CPU Time
f2	16	1.667	9.000	0.070s
rd53	32	1.333	11.967	0.126s
rd73	128	1.233	34.733	0.703s
sao2	1024	1.133	37.700	2.374s
bw	32	1.133	35.900	0.716s
rd84	256	1.100	58.633	2.232s
9sym	512	1.233	56.933	3.117s
c432	1000	1.267	40.067	6.791s
c499	1000	3.100	40.067	9.877s
c880	1000	1.567	82.667	19.078s
c1355	1000	1.633	95.267	81.864s
c1908	1000	14.933	553.200	220.589s
c2670	1000	6.333	573.733	463.418s
c3540	1000	10.167	867.067	1002.394s
c5315	1000	5.900	875.000	3257.420s
c7552	1000	7.933	1476.833	9368.553s

Table 4.4: Single Fault Algorithms Using Single Fault Patterns

I_{DDQ} test results to those produced by each faulty CUT. Obviously, the resolution is very high for these circuits. For ISCAS circuits like c1908 and c3540, there are a larger number of faults in F_L (15 and 11 respectively). However, the averages are still small (less than 20), so the resolution is still very good.

In order to develop a better appreciation for the resolution provided by the algorithm, consider the following. For a circuit containing n nodes, there are $2n$ stuck-at faults and $C_2^n = n!/(2!(n-2)!) \approx n^2$ two-node bridging faults. Obviously, for I_{DDQ}-based diagnosis (bridging fault diagnosis), there is a far greater number of faults than for voltage-based diagnosis (stuck-at fault diagnosis). Moreover, voltage-based diagnosis techniques can use information from a large number of primary outputs, while I_{DDQ}-based diagnosis has only one observation point. This suggests that many bridging faults will produce identical I_{DDQ} test results (i.e more aliasing will take place than during voltage-based diagnosis). However, from the small fault totals in table 4.4 (e.g. 15 for c1908 and 11 for c3540), the effectiveness of the single fault algorithm is apparent.

The size of fault set F_S can be relatively large (i.e. 1477 for c7552) for large circuits. However, when compared to the original number of faults in the entire circuit (i.e. 6,913,621 for c7552), the number of faults in F_S for each of the benchmarks is greatly

reduced. Moreover, the number of faults appearing in F_S is a secondary consideration to the number of faults in F_L since the fault in the CUT has a higher probability of being in F_L than being in F_S.

Comparative Simulations

The second set of simulations was performed on the ISCAS85 circuits using the test lengths specified in [14] instead of our standard test length of 1000 vectors. By using these test lengths for simulations, our results can be compared to the results obtained by the algorithm in [14] by Chakravarty. Due to the improvement in the analysis of feedback faults, it is expected that a higher resolution will be obtained using our algorithm.

Table 4.5 shows the results of these simulations. The table presents the test length for each benchmark, the average number of faults diagnosed using both our algorithm and the algorithm in [14], and the percentage improvement that is achieved by our algorithm. The results for both algorithms are averages of 25 simulations.

Circuit	Test Length	Our Algorithm			Chak.Algorithm	% of Improvement
		F_L	F_S	Total	Total	$\frac{Our Total}{Chak. Total} \times 100$
c432	130	1.240	41.280	43	87	49.4%
c499	425	2.480	40.120	43	570	7.6%
c880	80	20.040	135.240	156	172	90.7%
c1355	520	6.520	144.280	151	359	42.1%
c1908	865	23.040	559.960	583	952	61.2%
c2670	80	8.200	604.160	613	735	83.4%
c3540	450	12.120	902.920	916	1389	65.9%
c5315	690	6.160	875.040	882	5351	16.5%
c7552	90	6.360	1517.800	1525	1737	87.8%

Table 4.5: Results for Comparative Simulations

By comparing the values in the "Total" columns in table 4.5, it can be seen that a significant improvement is obtained using our algorithm. The last column in the table is used to illustrate the improvement in resolution provided by our algorithm. The numbers show how large our total fault set ($F_L + F_S$) is relative to Chakravarty's total fault set. For example, for c432, our fault set is 49.4% the size of Chakravarty's fault set ($43/87 \times 100 = 49.4\%$). Therefore, less than half as many faults must be examined using subsequent diagnosis methods for circuit c432 when our diagnosis

algorithm is used.

If subsequent diagnostic methods such as fault simulation are used to further diagnosis a circuit, a large amount of time and computer resources will be saved if our algorithm is used to provide the initial fault set instead of Chakravarty's algorithm. Although Chakravarty's algorithm takes less time to run than our algorithm, the time saved during subsequent diagnosis analysis allows our algorithm to be more useful when used as part of a complete diagnosis system.

The improvement in resolution achieved by our algorithm is attributed to the fact that it eliminates more feedback faults. For example, for benchmark c432, our algorithm identifies an average of 43 faults that can account for AP_{CUT}. Chakravarty's algorithm identifies an average of 87 faults. The difference of 44 faults is comprised solely of feedback bridging faults not eliminated by Chakravarty's algorithm.

Limitations of the Single Fault Assumption

The single stuck-at fault (SSF) model has been proven to be sufficient and effective for voltage testing applications. Statistics show that most faulty integrated circuits contain defects which can be modelled as SSFs, and that a large percentage of multiple faults can also be detected by SSF test sets. The corresponding statistics for I_{DDQ} testing are not available. The experiments presented in this subsection are designed to investigate the limitations of the single fault assumption, i.e. if a CUT contains more than one fault, the single fault algorithm may not be able to diagnose any of the multiple faults, or it may produce a misdiagnosis by reporting the presence of a single fault (aliasing).

To investigate these potential problems, 10 simulations were performed on each benchmark using the double fault I_{DDQ} test results discussed earlier. Any faults appearing in F_L for these simulations show that two or more faults can be improperly diagnosed as a single fault (aliasing). If set F_L contains no faults for these simulations, it is shown that the single fault algorithm is not able to locate the faults present in the CUT.

The results of the simulations using double fault patterns are summarized in table 4.6. For each circuit, F_L, F_S, and the CPU time are given. The numbers in these columns are averages over 10 simulations.

Circuit	Test Length	F_L	F_S	CPU Time
f2	16	0	8.900	0.073s
rd53	32	0	10.100	0.123s
rd73	[illegible]	0	32.600	0.652s
sao2	[illegible]24	0	36.300	2.117s
bw	32	0.200	32.700	0.630s
rd84	256	0	55.100	2.087s
9sym	512	0	53.900	3.045s
c432	1000	0	40.100	6.163s
c499	1000	0	40.000	8.545s
c880	1000	0	82.400	18.110s
c1355	1000	0	72.300	65.778s
c1908	1000	0	548.300	203.860s
c2670	1000	0	573.000	451.320s
c3540	1000	0	855.500	964.083s
c5315	1000	0	875.000	3196.940s
c7552	1000	0	1476.200	9051.060s

Table 4.6: Single Fault Algorithm Using Double Fault Patterns

From the values in column F_L in table 4.6, it can be seen that aliasing does not appear to be a problem for the single fault algorithm when double faults exist in the CUT. For all but one of the benchmarks (bw), no single faults exist which produce identical I_{DDQ} test results to those made from the randomly chosen double faults (i.e. no faults appear in set F_L). However, these results do illustrate that the single fault algorithm is not able to provide any useful diagnostic information for circuits containing two bridging faults, as neither of the single faults show up in set F_L.

Further conclusions can be drawn about the single fault algorithm by examining the total number of faults in column F_S for both table 4.4 and table 4.6. For all of the benchmarks, these totals are nearly equal. This suggests that for each circuit there is a constant number of sensitized feedback faults (e.g. a bridge between the input and output nodes of an inverter) which cannot be eliminated by the algorithm and are likely not responsible for producing AP_{CUT}. This result supports the claim that little useful information is generated by the single fault algorithm when more than one fault exists in the circuit.

4.3.3 Simulations Using the Multiple Fault Algorithm

Two different sets of simulations were performed to test the multiple fault algorithm. A set of simulations using single fault I_{DDQ} test results was performed to investigate how many double faults would produce I_{DDQ} test results identical to those of single faults. In this manner, the possibility of aliasing was examined. The second set of simulations was performed using double fault I_{DDQ} test results. These simulations help to analyse the diagnostic resolution obtained by the multiple fault algorithm. Tables containing the results of the multiple fault algorithm simulations are found in appendix D. All simulations were performed on a SPARC 20 workstation having 64MBytes of RAM.

Single Fault Simulations

The multiple fault algorithm, using a double fault assumption, was invoked using the same single fault I_{DDQ} test data as was used for the single fault algorithm simulations. These simulations attempt to provide evidence for the following claims.

- Circuits containing single faults are diagnosed correctly (i.e. they achieve the same results as the single fault algorithm).
- Aliasing can exist between single faults and multiple (double) faults.

Circuit	Test Length	F_L	F_P	F_S	F_U	$F_N(2)$		CPU Time
						F_{NS}	F_{NP}	
f2	16	1.677	18.633	18.633	0	91.600	18.167	0.133s
rd53	32	1.333	10.833	52.400	0	204.433	13.267	0.266s
rd73	128	1.233	44.733	114.667	0	469.233	7.300	1.886s
sao2	1024	1.133	52.900	78.133	0	286.233	15.167	4.100s
bw	32	1.133	243.800	135.700	0	3751.467	413.767	7.661s
rd84	256	1.100	452.500	219.900	1.000	4147.300	380.867	97.034s
9sym	512	1.233	27.500	132.133	0	924.433	9.600	4.895s
c432	1000	1.267	2.300	105.833	8.000	215.833	0.867	6.396s
c499	1000	3.100	63.667	1649.033	81.000	1341.067	66.767	172.121s
c880	1000	1.567	15.067	443.100	75.000	1486.600	1.500	29.139s
c1355	1000	1.633	183.733	2984.967	122.000	129442.333	28.400	618.337s
c1908	1000	14.933	451.033	5245.300	1085.000	90862.100	153.867	1937.839s
c2670	1000	6.333	42.267	2783.733	954.000	14648.467	274.967	658.179s
c3540	1000	10.167	86.133	3901.133	2201.000	88850.033	16.767	1327.839s
c5315	1000	5.900	4.700	6293.833	1461.000	39732.200	0	3526.281s
c7552	1000	7.933	805.067	12856.633	3199.000	262021.533	332.900	14657.493s

Table 4.7: Multiple Fault Algorithm Using Single Fault Patterns

Table 4.7 summarizes the simulation results. All of the totals in each column are averages over all 30 diagnosis runs. It can be observed that the numbers for F_L in table 4.7 (multiple fault algorithm) are the same as those in table 4.4 (single fault algorithm). This is because the single fault algorithm is a special case of the multiple fault algorithm and single faults will be diagnosed correctly using the either algorithm. However, due to the extra analysis performed by the multiple fault algorithm, it requires more CPU time than the single fault algorithm.

The totals in column F_S in table 4.7 are larger than the totals for F_S in table 4.4. This occurs because the multiple fault algorithm does not eliminate any faults when the input test vector activates an I_{DDQ} fault in the CUT. Consequently, the faults in F_S in table 4.4 are a subset of the faults in F_S in table 4.7. Although the resolution provided by the multiple fault algorithm for single faults is not as good for set F_S as it is for the single fault algorithm, no single faults identified by the single fault algorithm are eliminated by the multiple fault algorithm.

The data in table 4.7 also illustrates the possibility of aliasing. For these simulations, any double faults in fault sets F_{NP} and F_{NS} do not exist in the CUT since the I_{DDQ} test results are generated from circuits containing single faults. However, the multiple fault algorithm cannot determine if the actual fault(s) in the CUT will be found in the single fault sets (F_L and F_S) or the multiple fault sets (F_{NP} and F_{NS}). For example, for benchmark f2, either a single fault or a double fault may exist in the circuit. To determine the actual fault existing in the circuit, further diagnosis using another method must be employed.

Double Fault Simulations

A second set of simulations was performed using the 10 sample I_{DDQ} test results for double faults that are generated for each benchmark. These simulations illustrate the resolution provided by the multiple fault algorithm according to the number of double faults in sets F_{NP} and F_{NS}.

Table 4.8 summarizes the results of the double fault simulations on each benchmark. For all of the benchmarks (with the exception of bw), there are no faults in F_L. This indicates that there is more than one fault present in each of the benchmarks. F_{NP} and F_{NS} show the number of different double faults that can exist in

each benchmark.

Circuit	Test Length	F_L	F_P	F_S	F_U	$F_N(2)$		CPU Time
						F_{NS}	F_{NP}	
f2	16	0	51.300	25.400	0	229.300	146.300	0.285s
rd53	32	0	18.800	55.000	0	197.600	3.900	0.282s
rd73	128	0	69.200	119.600	0	484.300	2.600	1.995s
sao2	1024	0	45.100	72.700	0	214.300	6.500	3.222s
bw	32	0.200	960.000	187.200	0	12636.700	4272.100	76.067s
rd84	256	0	59.300	196.900	1.000	1366.100	4.600	4.852s
9sym	512	0	22.800	136.800	0	965.800	1.900	4.352s
c432	1000	0	7.900	109.700	8.000	224.000	2.100	5.967s
c499	1000	0	224.800	1644.900	81.000	1081.100	316.500	174.692s
c880	1000	0	50.300	461.800	75.000	1759.500	2.100	28.038s
c1355	1000	0	413.300	3010.400	122.000	117574.000	6.100	641.535s
c1908	1000	0	1177.300	5330.600	1085.000	102240.700	2051.400	2160.720s
c2670	1000	0	71.300	2790.100	954.000	12750.000	30.900	563.298s
c3540	1000	0	171.300	3962.600	2201.000	93612.500	81.500	1213.747s
c5315	1000	0	21.100	6303.300	1461.000	38626.800	27.400	2831.270s
c7552	1000	0	313.300	12824.500	3199.000	200954.900	74.300	11235.980s

Table 4.8: Multiple Fault Algorithm Using Double Fault Patterns

Although the numbers in these sets may seem too large to be of any use for diagnosis, they provide valuable information. First, the most probable double faults should be found in set F_{NP}. These faults are most likely to be the ones found in the CUT because they do not depend upon the pessimistic feedback fault assumptions used by the multiple fault algorithm. It can also be seen from table 4.8 that the number of faults in set F_{NP} is normally much smaller than the number in set F_{NS}. For example, for benchmark c7552, there are on average approximately 74 double faults which are diagnosed to set F_{NP} by the algorithm, as compared to over 200,000 faults in set F_{NS}. It will take a much shorter amount of time to analyse the faults in F_{NP} and, in most cases, the double fault will be found after this analysis. In this case, the large number of faults in set F_{NS} will not require further analysis.

It should also be noted that even though a large number of multiple faults can appear in set F_{NS}, the number is a very small fraction of the total number of multiple faults that can exist in the CUT. For example, for benchmark f2, there is a total number of $C_2^{378} = 71,253$ double faults, which is reduced to an average of approximately 230 faults in set F_{NS}, and 147 faults in set F_{NP}.

Remarks on the Multiple Fault Algorithm

Although the multiple fault algorithm is capable of correctly diagnosing single faults, extra CPU time is usually required to execute a diagnosis run. For the double fault patterns and the ISCAS benchmark circuits, the time required to execute the multiple fault algorithm (relative to the single fault algorithm) ranged from 1 (for c432) to 20 (for c499) times as long. Circuit c432 is the only circuit which ran faster using the multiple fault algorithm. However, since other processes were running on the computers during the times that simulations were being performed, the CPU time required for simulations may be slightly misleading for this circuit. For all larger circuits, the multiple fault algorithm takes considerably more CPU time than the single fault algorithm.

Due to the increase in CPU time, we recommend using the multiple fault algorithm and employing a double fault assumption only if the single fault algorithm fails to locate any single faults in the CUT. If a double fault assumption does not appear to be sufficient, a triple fault assumption can be used, and so on.

Information regarding the layout and routing of the CUT may also be used to improve the speed and resolution of the multiple fault algorithm. Most combinational circuits designed today contain many more gates than the Berkeley and ISCAS85 benchmarks. For these large circuits, the total number of faults can be extremely large and the time and computer memory required for multiple fault diagnosis may be infeasible. One solution to this problem is to physically partition the CUT into smaller modules and perform I_{DDQ} testing and diagnosis on each module. Another solution is to use information from the circuit layout. From an extracted circuit layout, it is possible to determine which nets in the circuit are routed close enough to one another to allow for a possible bridge to exist between them. This information can be used to construct a reduced initial fault set which can be passed to the multiple fault algorithm. Since the set will be much smaller than the exhaustive set of faults, a large reduction in testing time and computer memory requirements can be expected. As most computer-aided design (CAD) systems contain layout extraction tools, it is a relatively simple task to obtain the layout information for a particular CUT.

Another potential problem with the multiple fault algorithm is that the informa-

tion it produces can be affected by *fault masking*. Fault masking, as it relates to bridging faults, exists when the effects of one bridging fault overrides or otherwise interferes with the activation of a subsequent bridging fault. The effects of fault masking are most evident for feedback bridging faults.

As is discussed in chapter 2, feedback bridging faults may not produce faulty I_{DDQ} levels when the are activated. Such faults are not detectable using basic I_{DDQ} testing methods. If a fault alters a logic level on the bridged nodes, there is a possibility that the changed logic levels will affect the activation of subsequent bridging faults. In this case, a subsequent fault which would normally be activated by the current input vector may no longer be activated due to the masking effect of the first fault. Therefore, although both faults are expected to be activated by the current test vector, it is possible that no faulty I_{DDQ} will exist in the CUT. Unfortunately, there is currently no way to guard against this potential problem.

Apart from the fault masking problem described above, there may also be multiple faults which cannot be diagnosed even when a faulty I_{DDQ} test is received from the CUT. Such a case can occur if a fault from set F_L or F_S exists in the CUT simultaneously with a fault(s) from set F_P. If every time that the fault from set F_P is activated coincides with a fault from F_L or F_S being activated, it cannot be determined using the algorithm that the fault from F_P is in the CUT. This is because the faulty I_{DDQ} levels from the other faults will always mask the I_{DDQ} produced due to the fault in F_P. Therefore, the diagnosis algorithms may indicate that only a single fault exists in the CUT, while in actual fact, many other faults may also be present in the CUT. There is no way to solve this problem using only the information from I_{DDQ} tests.

4.4 Implementation

This section discusses the implementation issues of BRIDGE. BRIDGE is a gate-level logic simulator designed for I_{DDQ} diagnosis of gate level bridging faults. BRIDGE is a two-valued logic simulator, i.e., only two logic values, logic 1 and logic 0, are used. BRIDGE utilizes an event-driven signal propagation method as described in [3, page 69]. A flowchart illustrating the functions performed by BRIDGE is given in

figure 4.4.

4.4.1 BRIDGE Classes

BRIDGE was developed in a modular structure using C++. The classes in BRIDGE can be divided into three categories: circuit-related classes, simulation-related classes, and miscellaneous classes. Circuit-related classes define the structures of the logic gates available for simulations, construct a representation of the CUT, and control the way that logic information is passed between the gates during simulations. Simulation-related classes deal with test pattern generation, creating and injecting bridging faults into the CUT, implementing the diagnosis algorithms, and producing the simulation and diagnosis outputs. The miscellaneous classes do not perform specific simulation or diagnosis operations, but are used by other classes in BRIDGE.

Circuit-Related Classes

There are three circuit-related classes in BRIDGE:

Gate Class

The Gate class is a generic structure used to represent the different logic gates in the CUT. Each different Gate object that is created contains three basic attributes. The first attribute is an array of pointers, which identify the input nodes for the Gate. The second attribute is a single pointer which specifies the Gate's output node. The final attribute is a unique identifying number which is used to distinguish between different Gates. The identifying number is related to the output node specified for the gate in the corresponding ISCAS85 netlist representation.

The Gate class is hierarchical in nature. Each sub-class in Gate corresponds to one of the primary logic gates available for use in BRIDGE. Figure 4.5 illustrates the structure of the Gate class.

Node Class

The Node class represents a single node in the circuit. It accepts as input a single number (address) from a Gate which corresponds to the Gate's output node. Each Node object can have multiple fanouts. The following information is stored in every Node object:

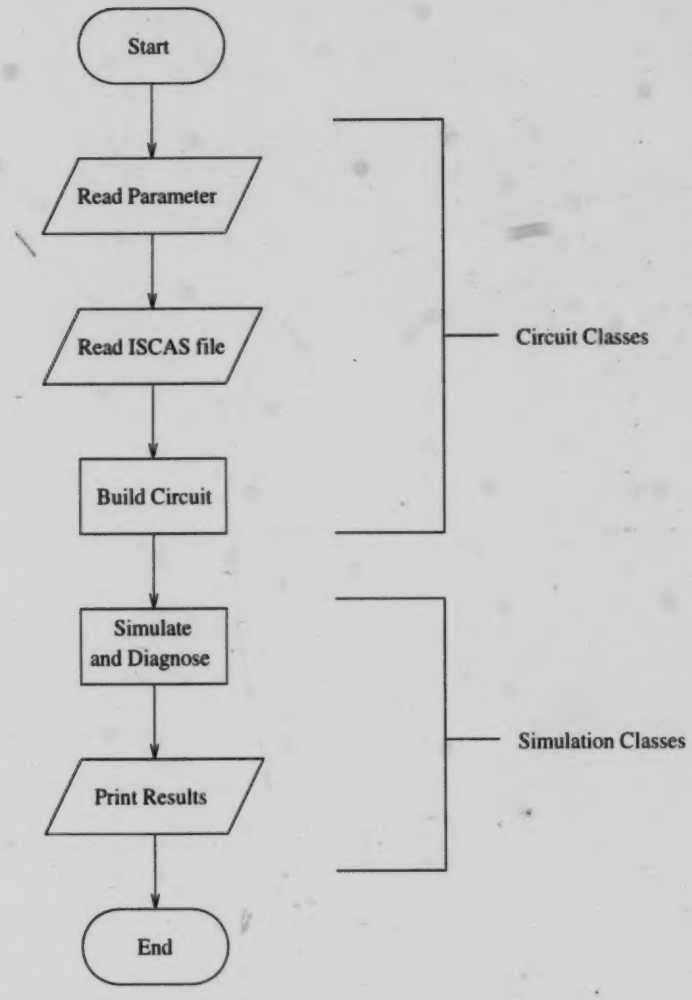


Figure 4.4: Flow Chart of BRIDGE

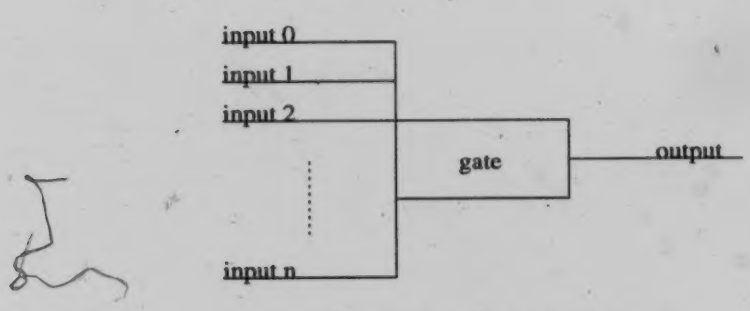


Figure 4.5: Gate Structure

1. **address:** A unique number differentiating all Nodes.
2. **name:** The name of the particular Node.
3. **value:** The logic value of the Node.
4. **last scheduled value and last scheduled time:** Values which keep track of the most recently scheduled value and the corresponding time at a Node.

A dynamic array (Page 81) called NodeArray is used to specify which numbers (addresses) correspond to existing Nodes in the circuit. NodeArray contains pointers to all of the Nodes. The pointers are indexed by the address of the Nodes. Since the addresses are not necessarily consecutive, NodeArray may contain gaps. If an address does not actually have an associated Node, the corresponding index in NodeArray is set to NULL. For example, if circuit Node number 3 in the ISCAS85 netlist represents a fanout node and is not directly associated with a logic gate, NodeArray[3] will contain a NULL value.

Signal Class

The Signal class represents the logic signal of a Node. Signal is implemented as a single byte: 1 for high, and 0 for low. High impedance and "don't care" logic values are not defined. All standard logical operations can be performed on Signal.

The Gate, Node and Signal classes combine to form the building block for the circuit being simulated. The building block is illustrated in figure 4.6.

Simulation-Related Classes

There are five different simulation-related classes:

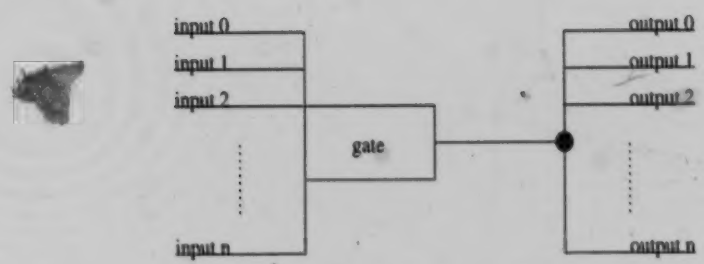


Figure 4.6: Circuit Building Block

TPG Class

All test pattern generators used by BRIDGE are defined in the TPG class. Each TPG object contains attributes for the initial state of the generator, the current state of the generator, and the number of bits in each test pattern. The TPG class is implemented as a state machine. Each machine can be iterated to its next state, or reset to its initial state. Within the TPG class are the BinaryCounter, LFSR, and LCAR subclasses. Each subclass uses different rules to compute the next test pattern provided to the CUT. A subclass called VectorList is employed if a set of test vectors stored in a text file are to be used as test patterns.

Fault Class

The Fault class is used construct a list of all the bridging faults in the CUT. The main component of a Fault object is a pair of pointers. Each pointer specifies the address of a Node in the CUT. Together the two pointers represent a bridging fault between the corresponding two Nodes. Each Fault object also contains the AP_{SIM_j} (expressed as a bit string), of the corresponding Fault, a flag specifying whether the Fault is a feedback fault, a non-feedback fault, or undetermined, and a flag specifying which fault set the Fault belongs to (i.e. F_L, F_P, F_S, F_{SE}, or F_U). All Fault objects are connected together in a linked list. Access to all Faults requires a pointer traversing from the first Fault to the last Fault.

MultiFault Class

The MultiFault class defines multiple faults of degree x, where x is supplied by the user. MultiFault is a list of the multiple fault sets, where each set contains pointers every single Fault in the set. As with the Fault class, Each MultiFault object is strung together in a linked list.

Event Class

The Event class controls the propagation of signals during a simulation. The structure of an Event list is shown in figure 4.7.

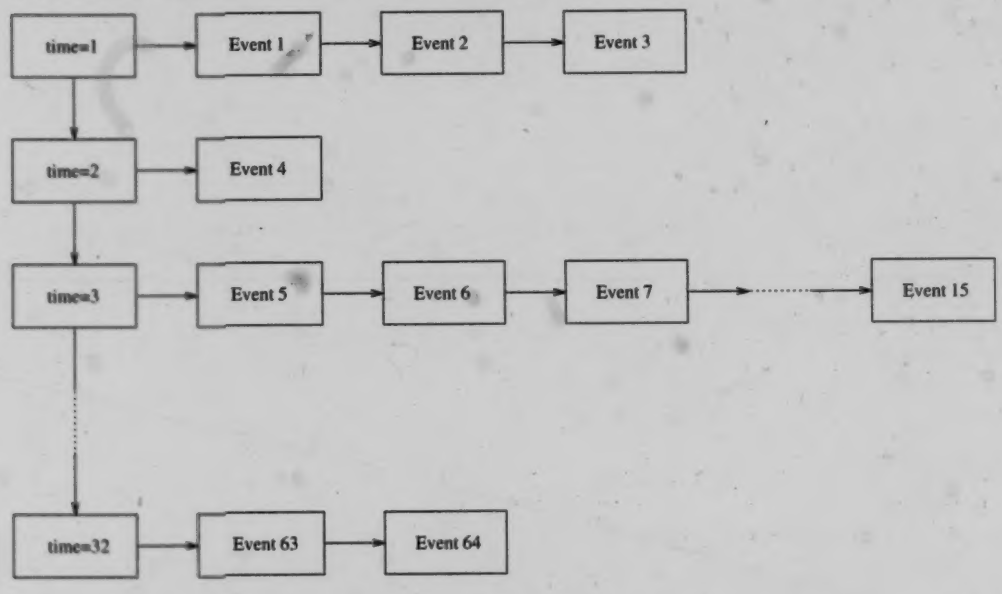


Figure 4.7: Event List Structure

Each Event object has two attributes: a Node address corresponding to the Event, and the Signal value that the Node should be changed to. For example, in figure 4.7, three Nodes are changed at time 1 while only one Node is changed at time 2.

As new test vectors are sent to the circuit inputs during simulation, new Events are scheduled at the input Nodes. The simulator retrieves the Events, updates the logic values on the effected Nodes, and schedules all required subsequent Events.

Sim Class

The Sim class contains routines for all the different types of simulations performed by BRIDGE. The Sim class uses the following routines during the execution of a diagnosis run.

- **ApplyVector** Receives the next state from the TPG and applies it to the primary inputs of the CUT.
- **Run** Interacts with the Event class to propagate the test vectors through the circuit.

- **Diagnosis** Executes a diagnosis run on the CUT based upon the AP_{CUT} given to the simulator.

- **Diagnosis (Faults Partitioned)** Executes a diagnosis on the circuit in smaller segments (i.e. smaller groups of faults are considered at a time instead of all the faults being considered at once).

Miscellaneous Classes

DynamicArray

DynamicArray is an array template used by many other classes in BRIDGE. Initially, DynamicArray consists of a pointer to an array of fixed size. If a large array is necessary, a new pointer-array set is created. Figure 4.8 illustrates the structure of DynamicArray.

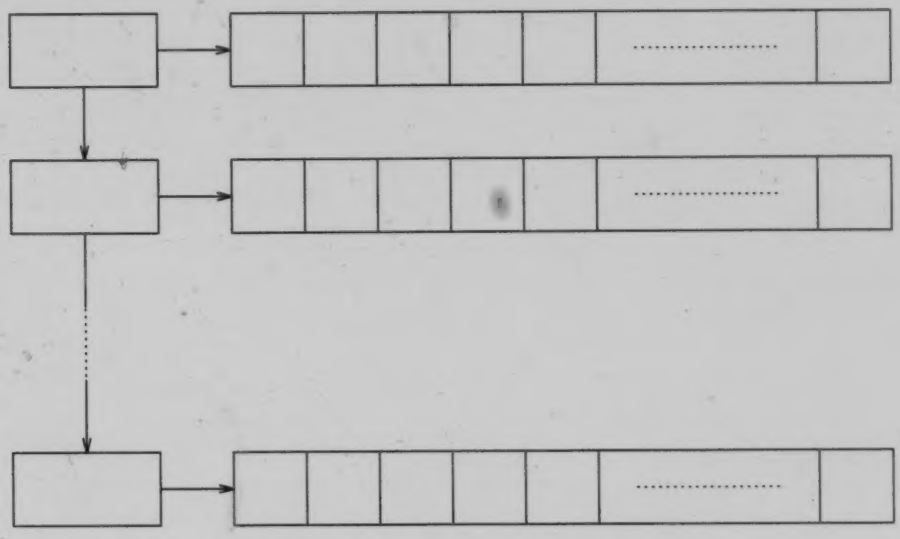


Figure 4.8: Structure of DynamicArray

4.4.2 Diagnosis Implementation

Once a circuit has been read into BRIDGE, diagnosis is carried out in two stages when the single fault algorithm is invoked, and four stages when the multiple fault algorithm is invoked. The first stage is used to eliminate faults from the initial fault set. This will help reduce the amount of memory required to perform a diagnosis. As

each test vector is applied to the CUT, BRIDGE uses simulation information and the corresponding bit value from AP_{CUT} (which is expressed as a bit string) to determine whether a fault should be eliminated or not. For the single fault algorithm, faults can be eliminated regardless of whether the bit from AP_{CUT} is a 1 or a 0. For the multiple fault algorithm, faults can only be eliminated if the bit from AP_{CUT} is a 0. As faults are eliminated during this step, the linked list of fault objects is updated.

During this stage, feedback faults are analysed to determine whether they are sensitized by particular test vectors. A sensitization check is performed by complementing the signal on the first node in the fault, and seeing if the change propagates down to the second node. If a feedback fault is sensitized by at least one test vector, the corresponding flag is set in the fault object.

The second stage of diagnosis is used to categorize the remaining faults. The categorization takes place after the entire value of AP_{SIM_j} has been recorded for each fault object in the linked list of faults. Faults are catagorized by comparing the string of bits in AP_{SIM_j} to the string of bits in AP_{CUT}.

When the single fault algorithm is employed, BRIDGE flags each fault object as belonging to either F_L or F_S. If $(AP_{SIM_j} = AP_{CUT})$, the object is flagged as belonging to F_L. If $(AP_{SIM_j} \neq AP_{CUT})$, the object is flagged as belonging to F_S.

When the multiple fault algorithm is invoked, each fault object belongs to one of F_L, F_P, F_S, or F_U. If $(AP_{SIM_j} = AP_{CUT})$, the fault is flagged as belonging to F_L. If $(AP_{SIM_j} \neq AP_{CUT})$, it belongs to either F_P or F_S. BRIDGE uses the sensitization flag in each fault object to decide whether the fault belongs in F_S or F_P. If the flag was set during stage 1, the fault is belongs to F_S. Otherwise, the fault belongs to F_P. Finally, if every entry in AP_{SIM_j} is 0, the fault is flagged as belonging to F_U.

The set of sensitized faults belonging to F_{SE} is also identified in this stage. For all faults in F_S, AP_{SIM_j} is logically ANDed to AP_{CUT}. If the result of the AND operation is equal to AP_{CUT}, the fault object is flagged as belonging to F_{SE} as well as F_S. After this stage, the single fault algorithm is completed.

The third stage of analysis (multiple fault algorithm only) is the implementation of procedure AP_UNIQUE. To implement this procedure, BRIDGE re-analyses the stored values of AP_{SIM_j} for fault objects in F_S and F_P. Each bit position in AP_{SIM_j} is analysed for each fault to see if it is a 1. If more than one fault has a 1 in a

particular bit position, the analysis stops, and proceeds to the next bit position. If any bit position is found for which only one fault has a value of 1, that fault object is flagged as belonging to F_L.

The final stage for the multiple fault algorithm is implemented using the MultiFault class. The MultiFault class implements the NTUPLE procedure described in chapter 3. MultiFault creates a new linked list of all diagnosed multiple faults. Each entry on the list contains pointers to the single faults which combine to create the multiple fault being considered. The new linked list is created by the following steps.

1. A multiple fault is created by pointing to x single faults from set F_P or $(F_S \backslash F_{SE})$. All possible single fault combinations from F_P and $(F_S \setminus F_{SE})$ are considered.

2. AP_{SIM_i} for all single faults are ORed together to form the AP_{SIM_n} of the multiple fault.

3. The new multiple fault AP_{SIM_n} is ANDed together with AP_{CUT}.

4. If the result of the AND gives a value equal to AP_{CUT}, the multiple fault is added to the new linked list.

After the above analysis is done, the new linked list contains all of the diagnosed multiple faults. Each entry on this list is also flagged as a sensitized fault (F_{NS}) if it contains at least one single fault from $(F_S \setminus F_{SE})$, or as an unsensitized fault (F_{NP}) if all the component single faults are from set F_P. Upon completion, BRIDGE prints out a summary page indicating the number of faults in each of the fault sets.

Currently, BRIDGE operates in a sequential fashion. However, a new version of the simulator, NEWBRIDGE, is under construction which analyses test vectors in parallel. Since logic simulation information can be generated far faster with a parallel simulator than it can be using a sequential simulator, it is expected that NEWBRIDGE will be considerably more efficient than BRIDGE.

Chapter 5

Conclusion

Gate-level bridging defects are a common type of defect in CMOS circuits [11, 19, 27, 38]. Due to the unpredictable effects that bridging defects have on circuit operation, testing and diagnosis methods employing voltage-based techniques are not always successful at detecting and locating bridging defects [18].

An important property of CMOS circuits is that very little current flows during the steady-state in a fault-free circuit. If a circuit contains a defect, an abnormally high level of quiescent current may flow during the application of certain test vectors. When gate-level bridges are activated in a CMOS circuit, a low resistance path is set up between Vdd and GND causing a high level of current to flow. Therefore, I_{DDQ} testing (steady-state current monitoring) is able to detect bridges regardless of whether they introduce erroneous voltages into the circuit.

For a circuit with n nodes, there are approximately n^2 two-node bridging faults that can exist in the circuit. Unlike voltage-based testing methods, I_{DDQ} tests use only one observation point to record information during testing. This makes it difficult to distinguish between different faults when I_{DDQ} tests are used for diagnostic purposes. This thesis has researched the problem of using the information from an I_{DDQ} test to diagnose gate-level bridges in a CMOS circuit.

The main contributions of this thesis are:

(1) Two new algorithms (a single fault algorithm and a multiple fault algorithm) to diagnose gate level bridging faults using information from logic simulations and I_{DDQ} tests are presented. Our single fault algorithm provides a better diagnosis (i.e. a higher diagnostic resolution) than a previously presented algorithm [14] since

we eliminate more feedback faults during each diagnosis session. Our multiple fault algorithm enables diagnosis to be performed on circuits containing more than one fault at any one time;

(2) A set of circuit design recommendations is given. Some of the recommendations, such as designing circuits using standard cell libraries, minimizing the number of inverters, and avoiding redundancies, help to increase the I_{DDQ} testability of a circuit which in turn helps to maximize the achievable diagnostic resolution. Other recommendations, such as re-ordering the test vectors applied to the circuit to eliminate faults quicker and finding test sets that activate each fault as uniquely as possible, will improve the performance of the algorithms;

(3) Extensive computer simulations are performed on Berkeley [41] and ISCAS85 [12] benchmarks to illustrate the merits and feasibility of the new algorithms.

The following conclusions are reached from the simulations performed for this thesis:

(1) Our single fault algorithm provides a significant improvement on diagnostic resolution than the algorithm presented in [14]. Our simulation results show that our algorithm produces diagnosed fault sets that are from 7.6% to 90.7% as large as the diagnosed sets produced by the algorithm in [14]. The smaller fault set size leads to a savings in the amount of CPU resources and time required to implement subsequent diagnostic methods such as circuit-level fault simulation;

(2) Our multiple fault algorithm is capable of correctly diagnosing all single faults as well as multiple faults. The multiple fault algorithm is used when both the single fault algorithm and subsequent diagnosis fail to locate any bridging faults. Aliasing (e.g. multiple faults produce the same I_{DDQ} test results as single faults) is unavoidable when using the multiple fault algorithm. When aliasing occurs, both single and double faults are placed in the diagnosed fault sets. Subsequent diagnostic methods, such as circuit-level fault simulation are required to determine which fault(s) is (are) actually in the circuit.

The result of this research suggests that the new diagnosis algorithms can be a valuable part of a complete bridging fault testing and diagnosis system. The following problems have been identified for further investigation:

(1) Different test sets can affect algorithm performance depending upon the test

set length and the "randomness" of the vectors in the set. Since I_{DDQ} tests require more time to execute than voltage-based tests using the same number of test vectors, the number of test vectors for each testing and diagnosis session should be minimized. A study to determine the test pattern generators which produce the shortest test sets that still enable a high diagnostic resolution will be useful in order to keep testing time to a minimum;

(2) Applying the algorithms to transistor-level faults. A previous study has shown that the transistor-level fault models and gate-level fault models overlap slightly, but usually they will test for different defects [7]. Moreover, it is more difficult to find test vectors to activate transistor-level nodes and activated transistor-level faults may not produce easily measurable levels of I_{DDQ}. Since the algorithms in this thesis employ a gate-level assumption, it is of interest to see whether they can be applied to transistor-level faults as well.

(3) BIST is commonly used in voltage-based testing architectures. Fault diagnosis techniques which utilize existing BIST resources are readily available [40, 45]. Exploring the possibility of combining both current-based and voltage-based diagnostic methods to achieve a higher quality of diagnosis is an interesting topic of future research.

Bibliography

[1] Jacob A. Abraham and W. Kent Fuchs. Fault and error models for VLSI. *Proceedings of the IEEE*, 74(5):639–654, 1986.

[2] Miron Abramovici and Melvin A. Breuer. Multiple fault diagnosis in combinational circuits based on an effect-cause analysis. *IEEE Transactions on Computers*, 29(6):451–460, 1980.

[3] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design.* Computer Science Press, 1990.

[4] John M. Acken and Steven D. Millman. Accurate modelling and simulation of bridging faults. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 17.4.1–17.4.4, 1991.

[5] R. C. Aitken and V. K. Agarwal. A diagnosis method using pseudo-random vectors without intermediate signatures. In *Proceedings of the International Conference on Computer Aided Design*, pages 574–577, 1989.

[6] Robert C. Aitken. Fault location with current monitoring. In *Proceedings of the IEEE International Test Conference*, pages 623–632, 1991.

[7] Robert C. Aitken. A comparison of defect models for fault location with Iddq measurements. In *Proceedings of the IEEE International Test Conference*, pages 778–787, 1992.

[8] Sara Baase. *Computer Algorithms Introduction to Design and Analysis.* Addison Wesley, second edition, 1988.

[9] Paul H. Bardell and William H. McAnney. Self-testing of multichip logic modules. In *Proceedings of the IEEE Test Conference*, pages 200–204, 1982.

[10] Paul H. Bardell, William H. Mcanney, and Jacob Savir. *Built-In Test for VLSI: Pseudorandom Techniques.* John Wiley & Sons, 1987.

[11] S. Wayne Bollinger and Scott F. Midkiff. On test generation for I_{DDQ} testing of bridging faults in CMOS circuits. In *Proceedings of the IEEE International Test Conference*, pages 598–607, 1991.

[12] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, 1985.

[13] Sreejit Chakravarty and Yiming Gong. An algorithm for diagnosing two-line bridging faults in combinational circuits. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 520–524, 1993.

[14] Sreejit Chakravarty and Minsheng Liu. Algorithms for I_{DDQ} measurement based diagnosis of bridging faults. *Journal of Electronic Testing: Theory and Applications*, 3(4):377–385, 1992.

[15] Sreejit Chakravarty and Sivaprakasam Suresh. I_{DDQ} measurement based diagnosis of bridging faults in full scan circuits. In *Proceedings of the 7th International Conference on VLSI Design*, pages 179–182, 1994.

[16] John C. Chan and Jacob A. Abraham. A study of faulty signatures using a matrix formulation. In *Proceedings of the IEEE International Test Conference*, pages 553–561, 1990.

[17] Henry Cox and Janusz Rajski. A method of fault analysis for test generation and fault diagnosis. *IEEE Transactions on Computer-Aided Design*, 7(7):813–833, 1988.

[18] F. Joel Ferguson, Martin Taylor, and Tracy Larrabee. Testing for parametric faults in static CMOS circuits. In *Proceedings of the IEEE International Test Conference*, pages 436–443, 1990.

[19] Hong Hao and Edward J. McCluskey. Resistive shorts within CMOS gates. In *Proceedings of the IEEE International Test Conference*, pages 292–301, 1991.

[20] E. Isern and J. Figueras. Analysis of I_{DDQ} detectable bridges in combinational CMOS circuits. In *Proceedings of the 12th IEEE VLSI Test Symposium*, pages 368–373, 1994.

[21] Kuen-Jong Lee and Melvin A. Breuer. Design and test rules for CMOS circuits to facilitate IDDQ testing of bridging faults. *IEEE Transactions on Computer-Aided Design*, 11(5):659–670, 1992.

[22] Terry Lee, Weitong Chuang, Ibrahim N. Hajj, and W. Kent Fuchs. Circuit level dictionaries of CMOS bridging faults. In *Proceedings of the 12th IEEE VLSI Test Symposium*, pages 386–391, 1994.

[23] Shu Lin and Jr. Daniel J. Costello. *Error Control Coding: Fundamental and Applications.* Prentice-Hall, 1983.

[24] Ding Lu and Carol Q. Tong. Feedback bridging fault detection using current monitoring. In *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, pages 1094–1097, 1992.

[25] Yashwant K. Malaiya, A. P. Jayasumana, and R. Rajsuman. A detailed examination of bridging faults. In *Proceedings of the International Conference on Computer Design*, pages 78–81, 1986.

[26] Weiwei Mao, Ravi K. Gulati, Deepak K. Goel, and Michael D. Ciletti. QUIETEST: A quiescent current testing methodology for detecting leakage faults. In *Proceedings of the International Conference on Computer-Aided Design*, pages 280–283, 1990.

[27] Peter C. Maxwell and Robert C. Aitken. Biased voting: A method for simulating CMOS bridging faults in the presence of variable gate logic thresholds. In *Proceedings of the IEEE International Test Conference*, pages 63–72, 1993.

[28] W. H. McAnney and J. Savir. There is information in faulty signatures. In *Proceedings of the IEEE International Test Conference*, pages 630–636, 1987.

[29] MCNC Center for Microelectronics. *OASIS Manual*, 1990.

[30] Jacob Millman and Arvin Grabel. *Microelectronics*. McGraw-Hill, second edition, 1987.

[31] Steven D. Millman, Edward J. McCluskey, and John M. Acken. Diagnosing CMOS bridging faults with stuck-at fault dictionaries. In *Proceedings of the IEEE International Test Conference*, pages 860–870, 1990.

[32] Alvin Poon. Bridge: A bridging fault simulator. Technical report, Department of Electrical and Computer Engineering, University of Alberta, 1996.

[33] Janusz Rajski and Jerzy Tyszer. On the diagnostic properties of linear feedback shift registers. *IEEE Transactions on Computer-Aided Design*, 10(10):1316–1322, 1991.

[34] Jeff Rearick and Janak H. Patel. Fast and accurate CMOS bridging fault simulation. In *Proceedings of the IEEE International Test Conference*, pages 54–62, 1993.

[35] M. Roca, E. Sicard, and A. Rubio. I_{DDQ} testing of oscillating bridging faults in CMOS combinational circuits. *IEE Proceedings*, 140(1):39–44, 1993.

[36] J. Savir and W. H. McAnney. Identification of failing tests with cycling registers. In *Proceedings of the IEEE International Test Conference*, pages 322–328, 1988.

[37] Jerry M. Soden, Charles F. Hawkins, Ravi K. Gulati, and Weiwei Mao. I_{DDQ} testing: A review. *Journal of Electronic Testing: Theory and Applications*, 3(4):291–303, 1992.

[38] Thomas M. Storey and Wojciech Maly. CMOS bridging fault detection. In *Proceedings of the IEEE International Test Conference*, pages 842–851, 1990.

[39] Charles E. Stroud and T. Raju Damarla. Improving the efficiency of error identification via signature analysis. In *Proceedings of the 13th IEEE VLSI Test Symposium*, pages 244–249, 1995.

[40] Xiaoling Sun. BISD-Scan: A built-in fault diagnosis scheme for scan-based VLSI circuits. Technical report, Department of Electrical and Computer Engineering, University of Alberta, 1996.

[41] Berkeley tools. Electrical Engineering and Computer Sciences Department, University of California at Berkeley.

[42] Gert-Jan Tromp and A. J. van de Goor. Logic synthesis of 100-percent testable logic networks. In *Proceedings of the IEEE International Conference of Computer Design*, pages 428–431, 1991.

[43] H. T. Vierhaus and U. Glaser W. Meyer. CMOS bridges and resistive transistor faults: IDDQ versus delay effects. In *Proceedings of the IEEE International Test Conference*, pages 83–91, 1993.

[44] John A. Waicukauski, Ved P. Gupta, and Sanjay T. Patel. Diagnosis of BIST failures by PPSFP simulation. In *Proceedings of the IEEE International Test Conference*, pages 480–484, 1987.

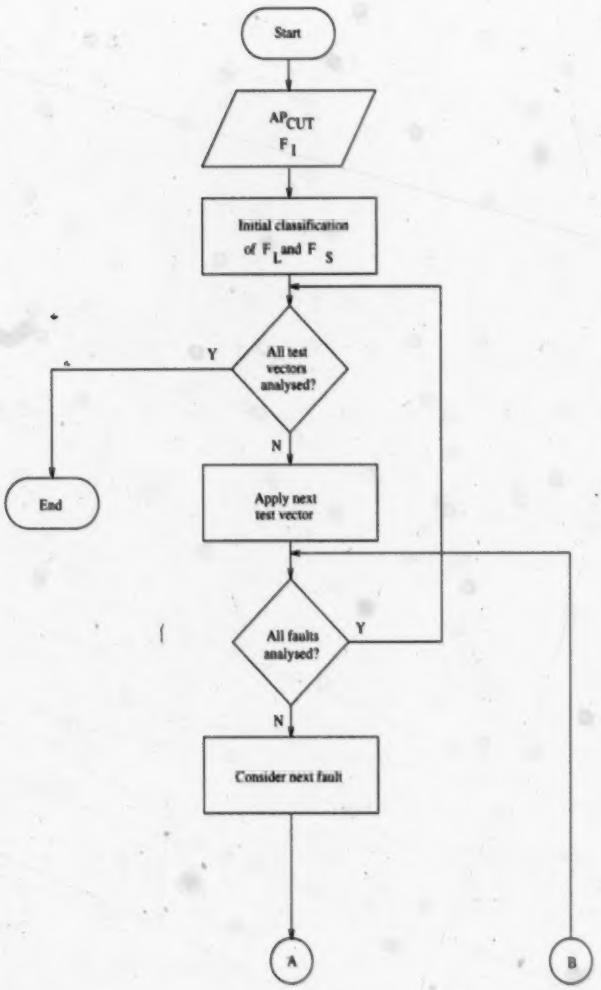
[45] Yuejian Wu and Saman M. I . Adham. BIST fault diagnosis in scan-based VLSI environments. To appear in *Proceedings of the IEEE International Test Conference*, 1996.

[46] Takao Yano and Hidetaka Okamoto. Fast fault diagnostic method using fault dictionary for electron beam tester. In *Proceedings of the IEEE International Test Conference*, pages 561–565, 1987.

Appendix A

Algorithm Flowcharts

A.1 Single Fault Algorithm Flowchart



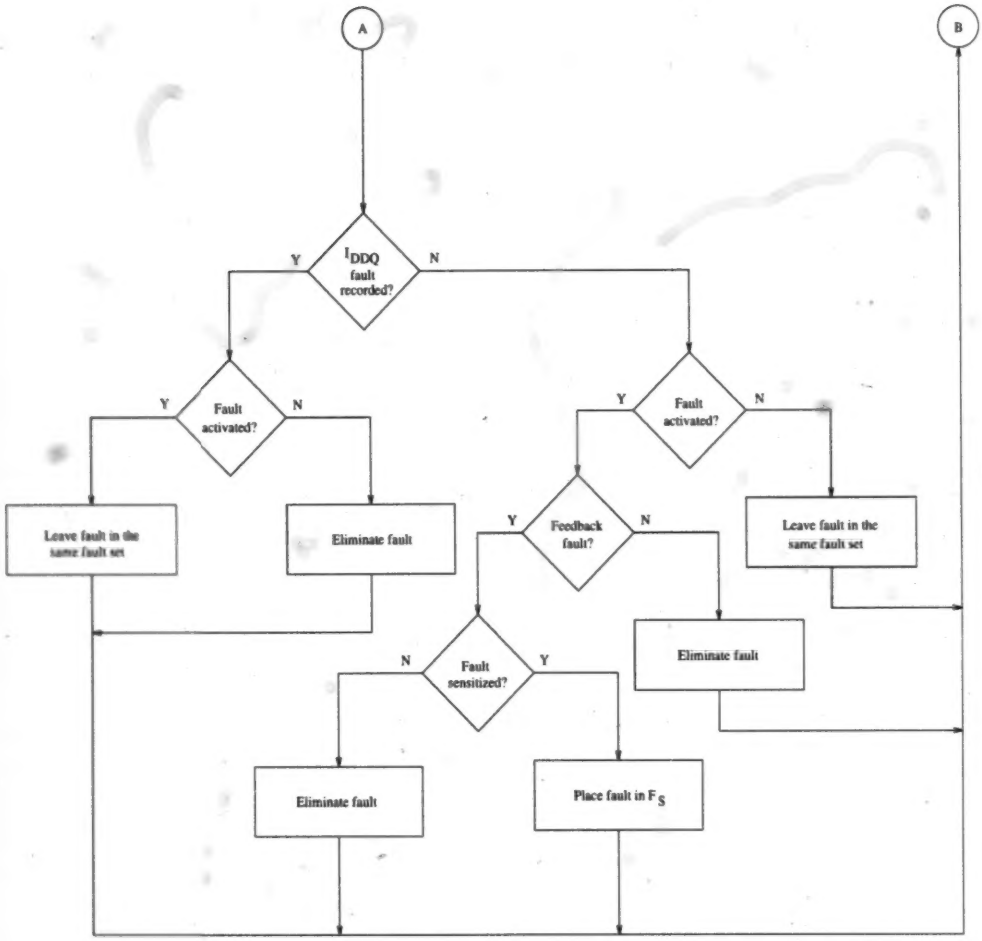
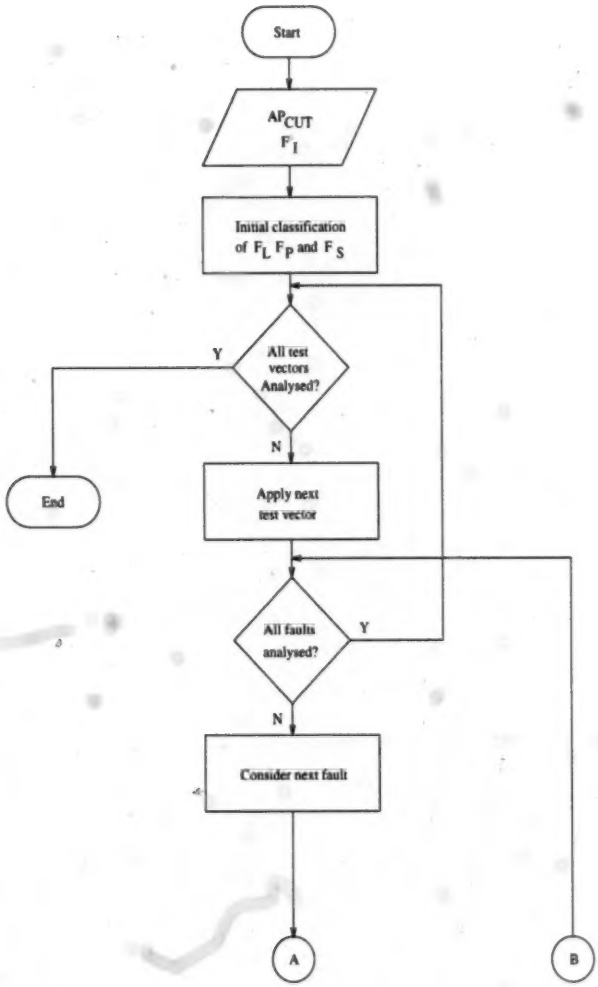


Figure A.1: Flowchart for Single Fault Algorithm

A.2 Multiple Fault Algorithm Flowcharts



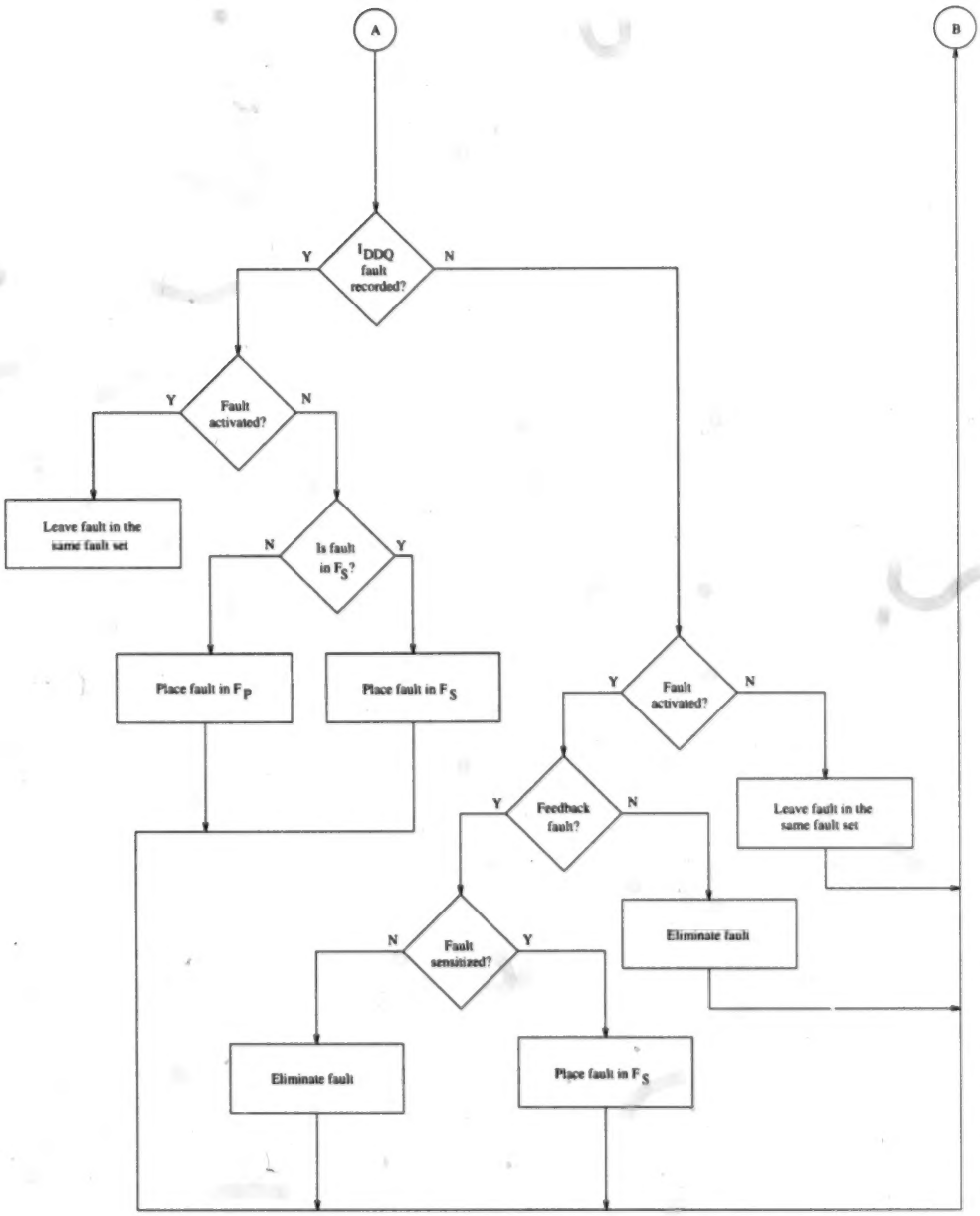


Figure A.2: Flowchart for Procedure CLASSIFY

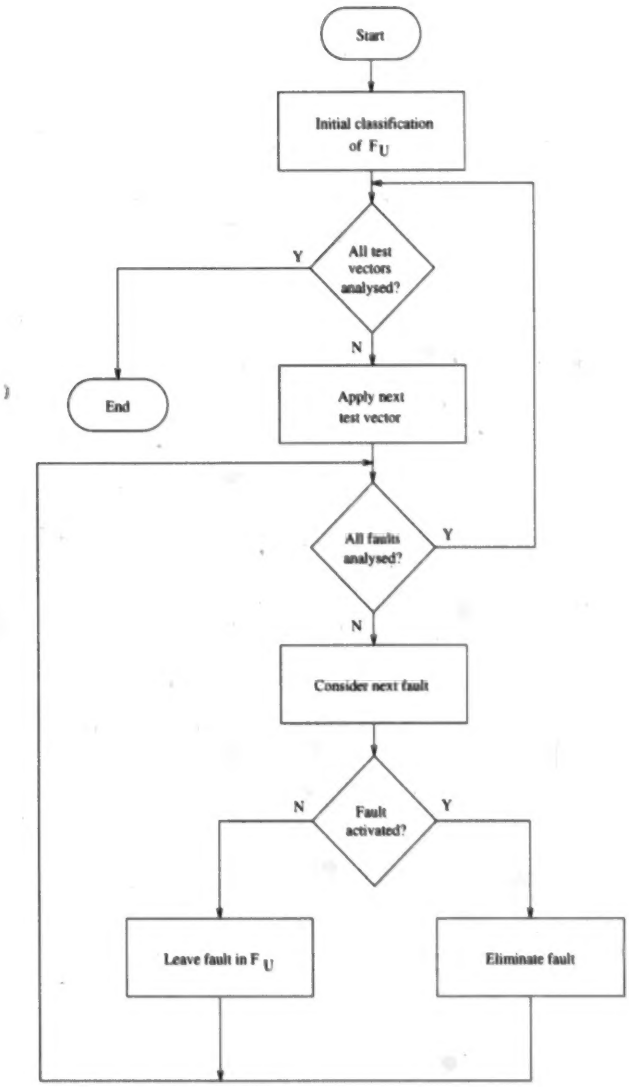


Figure A.3: Flowchart for Procedure UNDETECTED

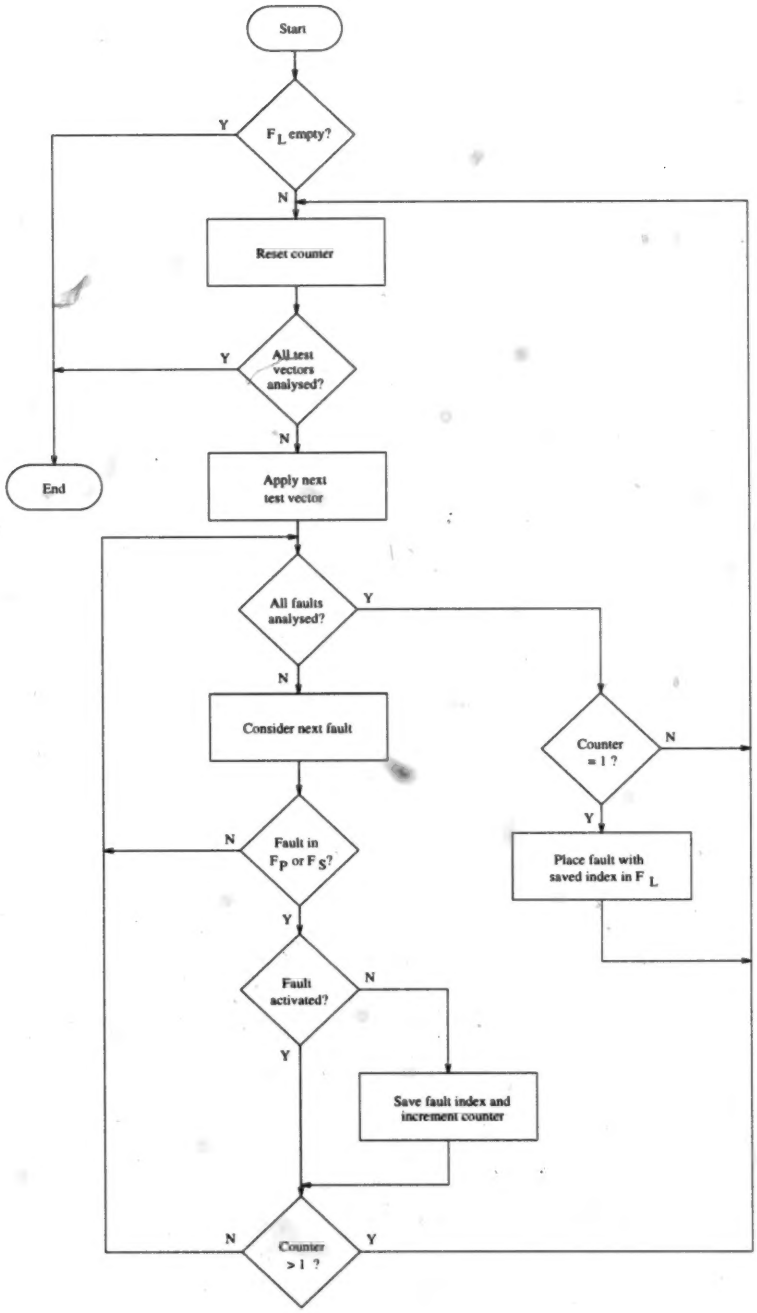


Figure A.4: Flowchart for Procedure AP_UNIQUE

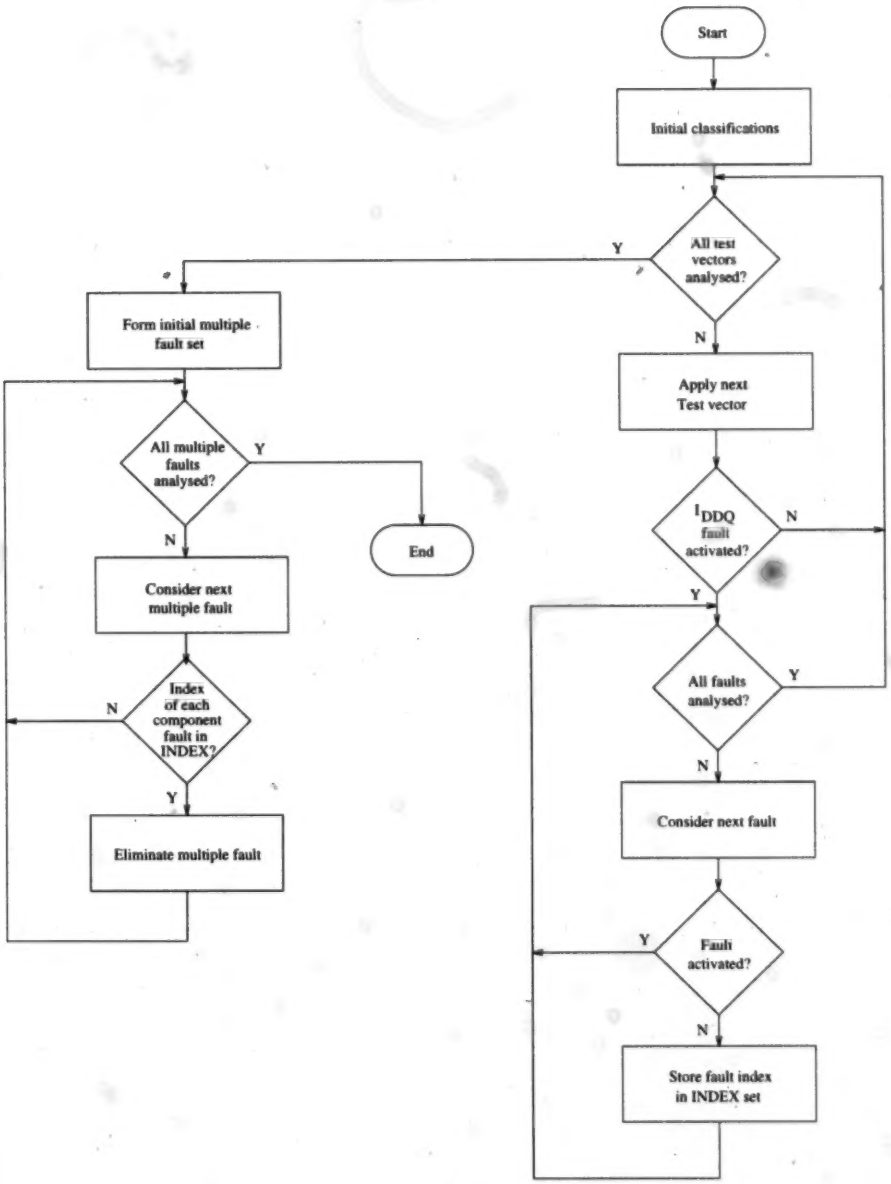


Figure A.5: Flowchart for Procedure NTUPLE

Appendix B

BRIDGE User's Manual

Michael Olson, Wes A. Tutak and Alvin S. Poon
Department of Electrical and Computer Engineering, University of Alberta

NAME

BRIDGE – a bridging fault simulator and diagnosis program

SYNOPSIS

br [options] circuit_file

DESCRIPTION

Reads in a description of a combinational circuit from a **circuit_file** in the ISCAS85 [12] netlist format. Simulates gate-level bridging faults for a given test pattern generator implemented by either a CA, LFSR, or Binary-Counter. The test patterns can also be from a file. **BRIDGE** produces the I_{DDQ} test results produced due to each bridging fault in a fault set.

Bridging fault diagnosis is performed if a sample set of I_{DDQ} test results are provided as input to **BRIDGE**. The output of the diagnosis is the total number of faults which are not eliminated during diagnosis. Bridge performs both single fault diagnosis, and multiple fault diagnosis, where a multiple fault is a combination of two or more single faults. The following information is produced by BRIDGE during a diagnosis session.

- Total number of faults considered.
- Total number of faults eliminated.

- Total number of *located* faults.
- Total number of *potential* faults (multiple fault diagnosis only).
- Total number of *sensitized* feedback faults.
- Total number of *multiple* faults. Total for both multiple faults consisting of only potential faults, and those containing at least one sensitized fault are given (multiple fault diagnosis only).

BRIDGE can also produce the SPICE format of the circuit netlist, or a truth table in the *espresso* format.

OPTIONS

1. TYPE OF SIMULATION

-ap Produce the I_{DDQ} test results as a binary sequence for a given set of input test vectors.

-av Produce the list of input test vectors, and the faults that each vector detects.

-tt Produce a truth table with the *espresso* format. the circuit under test.

-c Generate a minimum set of vectors that will test for all bridging faults in the circuit.

-df file Perform bridging fault diagnosis using the contents of file as the I_{DDQ} test results.

2. TYPE OF BRIDGING FAULTS TO CONSIDER

-nfbf Non-feedback bridging faults.

-fbf Feedback bridging faults.

-all Both non-feedback and feedback faults.

3. METHOD OF TEST PATTERN GENERATION

-bc Binary counter.

-lcar spec LCAR. spec is the LCAR specifications.

-lfsr poly LFSR. poly is the LFSR polynomial expressed in binary.

-tf file Test vectors read from file.

4. FAULT SET CONTROL

-ff file Initial fault list is contained in file.

-r n Initial fault list is a set of n randomly generated faults.

5. TEST VECTOR CONTROL

-i i Initial test vector, specified by the binary vector i.

-n n Total number of test vectors, specified by the decimal number n

6. DIAGNOSIS CONTROL

-dv n Print out the number of faults eliminated after n test vectors have been analysed. This can be specified up to 10 times.

-dsf List each fault (node pair) not eliminated during diagnosis.

-dap Display the corresponding I_{DDQ} test results for each test vector not eliminated during diagnosis.

-dn n Perform multiple fault diagnosis, assuming n faults occur simultaneously.

-dpf n Perform diagnosis on sections of n faults at a time until the entire fault list has been analysed.

-dem Perform single fault diagnosis only.

7. MISCELLANEOUS OPTIONS

-l Print the initial fault list.

-p Print out the list of circuit nodes.

-s Generate a netlist in the spice format.

-v Print our progress messages during execution.

-h Print out help summary.

Appendix C

Single Fault Algorithm Simulation Tables

C.1 Single Fault Diagnosis Using Single Fault Patterns

f2				
378 faults simulated 16 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	6	2	9	0.0667s
2	6	2	8	0.0667s
3	10	1	11	0.0833s
4	12	2	8	0.0833s
5	10	1	8	0.0667s
6	12	2	8	0.0500s
7	4	2	8	0.0833s
8	8	1	12	0.0667s
9	8	1	9	0.0833s
10	12	1	8	0.0667s
11	8	1	8	0.0500s
12	6	2	9	0.0667s
13	14	2	8	0.0667s
14	8	1	8	0.0667s
15	8	1	18	0.0667s
16	6	4	8	0.0833s
17	10	2	9	0.0667s
18	4	1	8	0.0833s
19	10	2	8	0.0833s
20	8	1	12	0.0667s
21	8	1	8	0.0833s
22	12	4	8	0.0500s
23	8	1	8	0.0667s
24	10	2	8	0.0667s
25	4	1	8	0.0667s
26	8	1	8	0.0667s
27	8	1	12	0.0667s
28	10	2	9	0.0667s
29	4	4	8	0.0833s
30	10	1	8	0.0667s
Average	8.400	1.667	9.000	0.070
Std. Dev.	2.603	0.907	2.098	0.010

Table C.1: Results for Benchmark f2

rd53				
1035 faults simulated 32 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	20	1	12	0.1333s
2	8	2	19	0.1167s
3	12	2	12	0.1167s
4	16	1	10	0.1167s
5	24	2	10	0.1167s
6	16	1	11	0.1333s
7	16	1	10	0.1333s
8	8	2	12	0.1333s
9	16	2	10	0.1333s
10	5	1	24	0.1667s
11	16	1	11	0.1167s
12	16	2	10	0.1500s
13	18	1	12	0.1333s
14	16	1	10	0.0833s
15	16	1	10	0.1333s
16	16	1	11	0.1167s
17	19	1	12	0.1333s
18	16	2	17	0.1333s
19	16	2	11	0.1333s
20	16	2	13	0.1333s
21	10	1	15	0.1333s
22	16	1	10	0.1167s
23	16	1	12	0.1333s
24	16	1	12	0.1500s
25	24	2	10	0.0833s
26	24	1	10	0.1167s
27	20	1	11	0.1333s
28	20	1	10	0.1000s
29	16	1	11	0.1167s
30	16	1	11	0.1167s
Average	16.133	1.333	11.967	0.126
Std. Dev.	4.295	0.471	3.049	0.017

Table C.2: Results for Benchmark rd53

rd73				
8778 faults simulated 128 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	64	1	33	0.7167s
2	92	1	34	0.6167s
3	66	1	35	0.7667s
4	80	1	33	0.6167s
5	82	1	35	0.6333s
6	78	1	32	0.6000s
7	70	1	32	0.7333s
8	56	1	32	0.6833s
9	104	1	33	0.7167s
10	64	3	33	0.6167s
11	72	1	34	0.6833s
12	64	1	32	0.6167s
13	46	1	36	0.7500s
14	24	1	38	0.7167s
15	65	1	32	0.6667s
16	38	1	32	0.6833s
17	48	1	34	0.7333s
18	36	1	37	0.7167s
19	96	1	33	0.6500s
20	64	1	34	0.6500s
21	64	2	35	0.6167s
22	64	2	35	0.6667s
23	88	2	34	0.7833s
24	106	1	33	0.6167s
25	48	2	33	0.7167s
26	112	1	32	0.6833s
27	64	1	34	0.9500s
28	64	1	32	0.6667s
29	2	1	66	1.0333s
30	30	2	34	0.7833s
Average	65.033	1.233	34.733	0.703
Std. Dev.	24.400	0.496	6.000	0.094

Table C.3: Results for Benchmark rd73

sao2				
12090 faults simulated 1024 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	512	1	37	2.2000s
2	976	2	36	1.8000s
3	512	1	36	2.5500s
4	1004	1	36	1.8167s
5	192	2	43	2.6333s
6	352	1	37	2.3333s
7	512	1	37	2.1333s
8	480	1	38	2.5333s
9	696	1	36	2.1333s
10	960	1	38	2.1167s
11	544	1	40	2.3833s
12	120	1	40	3.3833s
13	720	1	36	2.0333s
14	800	1	37	2.0833s
15	320	1	37	2.5333s
16	832	1	37	2.1667s
17	508	2	46	2.6333s
18	480	1	38	2.5000s
19	512	1	36	2.3833s
20	256	1	41	2.5667s
21	560	1	37	2.3000s
22	400	1	38	2.3667s
23	512	1	36	2.3833s
24	508	1	36	2.6167s
25	392	1	37	2.7667s
26	457	1	39	2.3833s
27	76	1	38	2.7667s
28	680	1	36	2.1000s
29	498	2	36	2.2667s
30	566	1	36	2.3667s
Average	531.233	1.133	37.700	2.374
Std. Dev.	227.325	0.340	2.283	0.308

Table C.4: Results for Benchmark sao2

bw				
12720 faults simulated 32 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	6	2	43	0.7000s
2	20	1	35	0.6333s
3	10	1	36	0.6000s
4	13	2	42	0.7833s
5	24	1	33	0.8500s
6	10	1	34	0.8167s
7	8	1	41	0.7667s
8	20	2	32	0.8667s
9	12	1	36	0.6167s
10	20	1	36	0.6000s
11	9	1	44	0.9167s
12	18	1	34	0.6333s
13	24	1	32	0.7167s
14	24	1	31	0.6500s
15	18	1	44	0.6833s
16	22	1	31	0.7000s
17	12	1	35	0.8000s
18	22	1	34	0.7333s
19	25	1	32	0.6333s
20	20	1	31	0.5833s
21	22	1	32	0.7667s
22	12	1	34	0.6500s
23	23	1	30	0.7833s
24	20	1	32	0.6833s
25	22	1	37	0.6333s
26	2	1	52	0.9500s
27	24	2	35	0.5667s
28	11	1	42	0.7833s
29	24	1	33	0.7333s
30	24	1	34	0.6333s
Average	17.367	1.133	35.900	0.716
Std. Dev.	6.463	0.340	5.042	0.099

Table C.5: Results for Benchmark bw

rd84				
25651 faults simulated 256 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	136	2	58	2.0333s
2	128	1	55	2.2167s
3	82	1	61	2.1833s
4	37	1	68	2.5833s
5	176	1	53	2.4500s
6	40	1	67	2.3000s
7	208	1	53	1.9500s
8	180	1	54	2.0167s
9	96	1	59	2.5167s
10	238	1	53	1.9333s
11	156	1	57	2.0833s
12	193	1	57	1.9667s
13	160	1	54	2.0333s
14	186	1	55	1.9667s
15	120	1	55	3.0167s
16	206	1	53	1.9500s
17	128	1	56	2.0667s
18	37	1	67	2.8833s
19	252	1	53	1.9667s
20	48	1	73	2.4667s
21	68	1	60	2.1167s
22	74	1	63	2.0167s
23	56	2	74	2.4167s
24	28	2	62	2.5667s
25	62	1	60	2.6833s
26	28	1	62	2.5500s
27	104	1	56	2.0000s
28	179	1	54	2.0000s
29	180	1	54	2.0667s
30	128	1	53	1.9667s
Average	123.800	1.100	58.633	2.232
Std. Dev.	65.764	0.300	5.936	0.300

Table C.6: Results for Benchmark rd84

9sym				
28441 faults simulated 512 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	256	1	55	2.8667s
2	130	1	61	2.9000s
3	160	1	56	3.2167s
4	256	1	53	2.9500s
5	122	1	60	3.6167s
6	48	2	62	3.4333s
7	288	1	55	2.8333s
8	256	1	55	2.7500s
9	320	1	57	2.5833s
10	256	1	53	2.7833s
11	352	1	55	2.7167s
12	256	1	54	3.4333s
13	256	1	53	3.0167s
14	406	2	53	2.5167s
15	365	1	55	2.6500s
16	188	1	53	3.2000s
17	256	1	55	2.8667s
18	88	1	63	3.7167s
19	280	1	53	3.1833s
20	76	1	63	3.8333s
21	92	1	60	3.7333s
22	56	1	67	3.4000s
23	160	1	56	3.1833s
24	52	2	56	3.9333s
25	352	2	54	2.7500s
26	448	2	55	2.6833s
27	256	2	56	2.9333s
28	220	2	66	4.2833s
29	256	1	56	2.8333s
30	389	1	58	2.7000s
Average	229.867	1.233	56.933	3.117
Std. Dev.	110.024	0.423	3.915	0.447

Table C.7: Results for Benchmark 9sym

c432				
19110 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	507	1	40	7.0833s
2	495	1	40	7.7333s
3	501	1	40	7.0667s
4	517	2	40	6.9000s
5	537	1	40	6.0833s
6	477	1	40	6.5833s
7	514	1	40	6.3833s
8	219	1	40	6.4667s
9	254	2	40	6.9667s
10	495	1	40	7.0000s
11	316	3	41	7.1667s
12	532	2	40	6.6000s
13	820	1	40	6.0167s
14	482	2	40	7.0333s
15	694	1	40	7.2833s
16	677	1	41	6.7833s
17	543	1	40	6.6833s
18	528	1	40	7.4167s
19	544	1	40	7.6167s
20	344	1	40	7.1167s
21	501	1	40	6.7667s
22	234	1	40	6.8333s
23	508	1	40	6.3167s
24	557	1	40	6.6500s
25	241	1	40	6.7167s
26	499	2	40	6.3333s
27	771	1	40	6.6333s
28	524	2	40	7.1167s
29	441	1	40	6.1667s
30	502	1	40	6.2000s
Average	492.467	1.267	40.067	6.791
Std. Dev.	140.758	0.512	0.249	0.430

Table C.8: Results for Benchmark c432

c499				
29403 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	498	1	40	8.9167s
2	496	1	40	9.6667s
3	496	2	40	9.5000s
4	510	5	40	10.1667s
5	498	1	40	11.6833s
6	491	1	40	9.6167s
7	492	2	40	8.5167s
8	518	5	40	8.3167s
9	487	1	40	10.1167s
10	496	1	40	10.2667s
11	509	1	40	10.5833s
12	483	5	40	10.1333s
13	501	1	40	9.6167s
14	480	1	40	9.9500s
15	480	1	40	9.8833s
16	506	1	40	9.4667s
17	482	5	40	10.9333s
18	498	1	41	8.4333s
19	501	5	40	9.0333s
20	494	5	40	9.4833s
21	499	5	40	10.3333s
22	490	1	40	8.9667s
23	503	1	40	10.7333s
24	508	1	41	8.9333s
25	512	1	40	10.6500s
26	502	26	40	9.2167s
27	486	5	40	9.3000s
28	512	5	40	10.0667s
29	509	1	40	12.5833s
30	511	1	40	11.2333s
Average	498.267	3.100	40.067	9.877
Std. Dev.	10.230	4.614	0.249	0.951

Table C.9: Results for Benchmark c499

c880				
97903 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	502	1	83	18.9667s
2	332	1	82	21.0167s
3	709	4	86	17.3333s
4	252	2	82	19.6833s
5	306	1	83	20.5333s
6	485	1	80	18.4833s
7	145	2	84	20.3333s
8	497	1	84	18.8167s
9	512	1	82	18.1833s
10	555	1	82	17.1333s
11	497	1	81	18.5167s
12	646	2	82	17.6167s
13	375	1	83	19.6500s
14	694	4	84	18.6333s
15	609	1	81	18.5333s
16	497	1	86	20.9167s
17	482	1	84	20.0333s
18	549	1	81	19.6167s
19	66	4	84	21.7167s
20	484	1	83	20.2500s
21	489	1	82	18.0667s
22	501	2	85	18.0333s
23	436	1	83	17.8167s
24	471	2	82	18.9500s
25	302	1	81	18.8167s
26	498	1	80	18.9667s
27	487	1	82	18.1833s
28	464	2	84	18.9167s
29	477	2	82	19.1833s
30	481	2	82	19.4333s
Average	460.000	1.567	82.667	19.078
Std. Dev.	138.674	0.920	1.513	1.108

Table C.10: Results for Benchmark c880

c1355				
171991 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	251	1	119	91.2833s
2	493	1	70	74.8833s
3	508	2	78	84.8000s
4	371	1	98	84.7000s
5	509	1	82	81.4000s
6	480	1	86	68.5667s
7	501	2	83	70.2167s
8	515	2	64	65.1833s
9	490	2	80	77.9333s
10	507	5	88	82.9167s
11	124	1	151	113.1167s
12	355	1	96	98.5500s
13	505	2	79	62.1000s
14	499	2	78	77.8833s
15	476	1	82	58.8667s
16	499	1	83	83.3000s
17	996	1	54	45.1833s
18	5	1	297	167.0000s
19	249	1	125	101.7833s
20	504	5	98	94.2333s
21	514	2	66	77.9333s
22	251	1	136	82.7333s
23	384	1	102	78.6833s
24	239	1	101	82.9333s
25	481	5	102	89.7000s
26	518	1	70	76.0333s
27	382	1	84	83.4333s
28	751	1	64	66.0667s
29	671	1	60	48.3333s
30	492	1	82	86.1667s
Average	450.667	1.633	95.267	81.864
Std. Dev.	180.217	1.197	43.310	21.304

Table C.11: Results for Benchmark c1355

c1908				
416328 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	502	5	555	226.2667s
2	484	8	568	230.5667s
3	507	2	552	202.0667s
4	514	24	549	220.7333s
5	515	8	554	216.1333s
6	499	8	556	207.2167s
7	515	3	549	245.4833s
8	521	1	556	204.2333s
9	526	2	549	241.4333s
10	480	15	550	203.9167s
11	509	6	546	193.5167s
12	524	22	560	218.8833s
13	491	6	553	215.4167s
14	506	119	547	228.2500s
15	504	12	550	196.9667s
16	516	26	550	206.4167s
17	497	21	551	233.4000s
18	479	6	550	200.2167s
19	491	18	559	228.9833s
20	446	5	553	218.4833s
21	507	6	552	231.6333s
22	477	25	562	230.3333s
23	483	39	549	243.6333s
24	502	6	550	226.6000s
25	736	1	559	203.9667s
26	483	5	552	219.5333s
27	511	33	551	214.9667s
28	490	2	558	252.0167s
29	481	2	555	225.8667s
30	521	12	551	230.5333s
Average	507.233	14.933	553.200	220.589
Std. Dev.	45.926	21.752	4.785	14.961

Table C.12: Results for Benchmark c1908

c2670				
1016025 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	463	6	573	455.0167s
2	163	4	589	487.7667s
3	489	14	573	450.4333s
4	541	3	573	457.9667s
5	662	1	573	439.3833s
6	494	1	573	467.2167s
7	480	2	573	459.0833s
8	529	6	573	456.9167s
9	548	8	573	448.8667s
10	495	18	573	458.6667s
11	472	2	573	472.9667s
12	516	6	573	479.6833s
13	493	2	573	454.6500s
14	239	2	573	468.0500s
15	432	4	579	477.8500s
16	564	5	573	464.3333s
17	470	1	573	481.4500s
18	503	1	573	476.9667s
19	489	2	573	475.3833s
20	473	2	573	455.6167s
21	456	20	573	483.7833s
22	816	2	573	446.1167s
23	514	6	573	462.0833s
24	486	2	573	452.9667s
25	482	7	573	475.2500s
26	176	4	573	479.4167s
27	476	21	573	473.6833s
28	780	24	573	438.1333s
29	534	2	573	461.1333s
30	523	12	573	441.7167s
Average	491.933	6.333	573.733	463.418
Std. Dev.	130.700	6.472	3.032	13.622

Table C.13: Results for Benchmark c2670

c3540				
1476621 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	467	8	853	1047.8833s
2	126	1	914	997.3667s
3	507	1	861	1037.7833s
4	568	9	849	977.9167s
5	513	16	868	963.5667s
6	347	2	853	1081.4500s
7	715	16	855	932.4333s
8	528	2	853	969.5667s
9	281	8	929	1024.8333s
10	396	1	861	1064.2500s
11	353	4	873	1070.3333s
12	516	1	857	976.2833s
13	506	18	852	1078.6833s
14	534	4	861	1011.1667s
15	518	2	855	1012.6167s
16	633	16	853	976.1333s
17	512	1	853	1040.1833s
18	558	24	872	1000.8333s
19	612	19	893	986.4167s
20	450	28	937	1000.6167s
21	742	1	855	944.7333s
22	243	24	872	1054.8667s
23	513	9	870	1034.4667s
24	825	51	859	948.2500s
25	335	1	866	1074.1500s
26	764	19	855	954.1833s
27	823	9	853	941.9333s
28	527	2	870	980.2833s
29	523	6	851	953.4167s
30	793	2	859	935.2167s
Average	524.267	10.167	867.067	1002.394
Std. Dev.	166.905	11.130	22.082	46.278

Table C.14: Results for Benchmark c3540

c5315				
3086370 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	423	1	875	3290.2167s
2	484	28	875	3193.6333s
3	463	6	875	3355.3167s
4	650	4	875	3231.6500s
5	502	2	875	3208.2167s
6	357	1	875	3243.6500s
7	475	3	875	3230.9167s
8	525	8	875	3230.4500s
9	514	22	875	3260.3500s
10	685	1	875	3258.2000s
11	511	3	875	3226.6833s
12	544	5	875	3184.6333s
13	463	15	875	3221.3333s
14	512	1	875	3300.4167s
15	539	4	875	3196.8167s
16	447	14	875	3308.9833s
17	484	4	876	3226.9667s
18	485	1	875	3202.9000s
19	491	2	875	3244.7667s
20	462	6	875	3300.1833s
21	284	2	875	3317.1000s
22	724	8	875	3213.1500s
23	480	4	875	3286.3000s
24	545	7	875	3350.1833s
25	275	2	875	3355.0667s
26	450	12	875	3296.7333s
27	475	6	875	3333.2167s
28	585	1	875	3222.2000s
29	528	2	875	3206.2167s
30	661	2	875	3226.1500s
Average	500.767	5.900	875.033	3257.420
Std. Dev.	97.126	6.353	0.180	50.886

Table C.15: Results for Benchmark c5315

c7552				
6913621 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	414	12	1477	9968.6667s
2	487	5	1477	9229.0167s
3	521	10	1477	8923.2833s
4	358	20	1481	9413.4167s
5	514	2	1477	9295.7167s
6	548	1	1475	9077.3667s
7	506	28	1478	9412.4667s
8	451	5	1475	9593.0500s
9	592	1	1475	9497.1000s
10	368	1	1481	9614.4833s
11	818	1	1475	9225.1000s
12	510	17	1477	9250.3667s
13	350	1	1475	9415.9500s
14	504	1	1477	9379.3333s
15	474	3	1477	9171.9000s
16	465	5	1475	9445.3167s
17	511	3	1477	9422.7667s
18	427	4	1475	9264.1000s
19	514	12	1477	9307.8667s
20	497	3	1475	9299.7000s
21	494	2	1475	9370.8667s
22	910	3	1475	9151.0833s
23	475	4	1475	9399.7333s
24	463	4	1477	9433.6167s
25	523	7	1477	9471.1667s
26	161	2	1485	9633.3333s
27	449	27	1479	9474.2167s
28	561	51	1475	9507.6833s
29	506	1	1475	9210.2333s
30	423	2	1479	9197.7000s
Average	493.133	7.933	1476.833	9368.553
Std. Dev.	127.377	10.853	2.282	193.303

Table C.16: Results for Benchmark c7552

C.2 Comparative Diagnosis Simulations

c432				
19110 faults simulated 130 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	65	1	40	2.3667s
2	69	1	40	2.0500s
3	58	1	40	2.0833s
4	68	2	40	2.2667s
5	64	1	40	1.4833s
6	57	1	40	1.9833s
7	66	1	40	1.4333s
8	22	1	42	1.6667s
9	26	2	50	2.2333s
10	68	1	40	2.2667s
11	37	3	41	2.1333s
12	65	2	40	1.9167s
13	113	1	41	1.3667s
14	67	2	40	2.1000s
15	94	1	40	1.8167s
16	83	1	41	1.4333s
17	62	1	40	2.0333s
18	69	1	40	1.8667s
19	63	1	40	2.2833s
20	29	1	40	2.3167s
21	67	1	40	2.1000s
22	33	1	41	1.8833s
23	66	1	40	1.5333s
24	76	1	41	2.0000s
25	28	1	55	1.8167s
Average	60.600	1.240	41.280	1.937
Std. Dev.	21.105	0.512	3.424	0.297

Table C.17: Results for Benchmark c432

c499				
29403 faults simulated 425 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	197	2	40	5.1667s
2	200	2	40	5.8667s
3	223	2	40	5.4333s
4	220	5	40	5.9167s
5	202	1	40	7.8000s
6	214	1	40	5.6167s
7	211	2	40	4.5000s
8	219	5	40	4.5500s
9	207	1	40	6.0667s
10	200	1	40	6.0333s
11	203	1	40	5.7667s
12	210	5	40	5.8500s
13	216	1	40	5.7167s
14	202	1	41	6.1333s
15	202	1	40	5.8333s
16	209	1	40	5.0667s
17	204	5	40	7.1333s
18	220	1	41	4.3167s
19	218	5	40	5.0000s
20	213	5	40	5.4333s
21	207	5	40	6.0167s
22	202	1	40	5.0333s
23	235	1	40	6.3667s
24	216	1	41	4.3333s
25	217	6	40	5.9500s
Average	210.680	2.480	40.120	5.636
Std. Dev.	8.952	1.857	0.325	0.796

Table C.18: Results for Benchmark c499

c880				
97903 faults simulated 80 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	38	1	121	5.8833s
2	29	1	139	6.2833s
3	74	95	133	4.2667s
4	13	29	174	6.5500s
5	23	1	123	5.9500s
6	33	2	116	5.4833s
7	3	67	204	7.0333s
8	39	3	123	5.2833s
9	39	1	117	5.3833s
10	54	1	126	4.3333s
11	27	1	127	5.9667s
12	42	40	118	5.1333s
13	15	1	127	6.9167s
14	34	4	118	5.9667s
15	63	1	120	4.1667s
16	29	1	124	7.3000s
17	41	1	116	6.7333s
18	54	1	117	5.9667s
19	2	136	280	8.8833s
20	31	1	121	6.9667s
21	44	1	120	4.3667s
22	55	2	121	4.2667s
23	48	1	135	4.1833s
24	49	2	120	5.5667s
25	19	107	141	5.8833s
Average	35.920	20.040	135.240	5.789
Std. Dev.	17.357	37.900	35.322	1.147

Table C.19: Results for Benchmark c880

c1355				
171991 faults simulated 520 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	126	1	144	59.7167s
2	257	3	117	56.9333s
3	255	2	126	65.4000s
4	186	1	124	54.5500s
5	254	1	122	55.0333s
6	256	1	121	46.0667s
7	249	8	124	46.2667s
8	283	2	104	48.3333s
9	258	2	116	55.0167s
10	266	5	114	60.5167s
11	67	1	165	75.8500s
12	187	1	120	67.0167s
13	262	44	129	44.3167s
14	256	38	146	66.7333s
15	233	1	122	42.4833s
16	256	34	121	64.4333s
17	517	1	89	29.0333s
18	3	1	551	119.5000s
19	129	1	161	71.1667s
20	279	5	122	67.3833s
21	270	2	106	60.9000s
22	131	1	152	50.7833s
23	191	1	144	54.6667s
24	138	1	133	60.9167s
25	253	5	134	64.8667s
Average	222.480	6.520	144.280	59.515
Std. Dev.	93.469	12.083	84.754	15.958

Table C.20: Results for Benchmark c1355

c1908				
416328 faults simulated 865 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	435	5	558	199.9333s
2	416	189	573	202.2333s
3	436	2	559	185.0333s
4	448	24	555	197.4167s
5	447	8	563	195.0333s
6	429	8	565	203.1333s
7	448	3	553	220.9667s
8	451	1	565	180.0000s
9	452	2	556	219.3500s
10	419	15	556	182.8667s
11	428	6	554	175.1833s
12	451	22	566	194.0167s
13	420	6	559	193.0667s
14	435	120	552	202.2000s
15	432	12	557	179.3000s
16	440	26	558	187.4167s
17	438	21	557	212.5167s
18	411	6	556	203.6333s
19	431	18	566	209.5000s
20	396	5	562	197.4167s
21	440	6	559	209.5167s
22	417	25	572	209.0500s
23	414	39	556	219.8833s
24	429	6	557	208.0333s
25	634	1	565	184.0500s
Average	439.880	23.040	559.960	198.830
Std. Dev.	42.047	41.106	5.473	12.944

Table C.21: Results for Benchmark c1908

c2670				
1016025 faults simulated 80 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	26	10	616	57.8667s
2	12	60	685	85.9167s
3	36	16	598	54.5833s
4	53	3	600	65.1833s
5	60	1	589	47.8833s
6	44	1	593	66.8167s
7	30	2	615	67.5167s
8	47	6	598	54.4333s
9	50	12	583	51.7833s
10	50	22	596	65.2833s
11	21	2	609	78.1167s
12	36	6	593	81.7333s
13	43	2	593	58.3167s
14	24	2	611	73.5833s
15	21	4	622	82.1167s
16	33	7	608	69.6167s
17	32	1	608	85.8667s
18	38	1	595	71.9667s
19	29	4	589	83.7167s
20	42	2	584	59.9167s
21	20	24	627	80.5833s
22	68	2	614	49.8833s
23	46	6	591	63.0333s
24	37	2	585	57.9667s
25	34	7	602	78.1167s
Average	37.280	8.200	604.160	67.672
Std. Dev.	13.046	12.261	20.303	11.746

Table C.22: Results for Benchmark c2670

c3540				
1476621 faults simulated 450 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	205	8	901	551.1667s
2	55	1	934	479.2167s
3	231	1	892	564.3667s
4	254	9	888	479.0000s
5	233	16	901	452.8667s
6	152	2	888	571.8667s
7	316	16	902	437.6333s
8	227	2	889	469.1333s
9	136	8	944	524.9833s
10	180	1	892	533.7833s
11	167	4	913	559.9167s
12	225	1	901	462.0500s
13	218	54	897	567.7000s
14	256	4	899	522.6000s
15	234	2	912	523.7333s
16	292	16	887	484.9167s
17	222	1	901	550.1333s
18	256	24	903	498.7167s
19	285	19	911	490.6833s
20	209	28	949	523.6500s
21	338	1	892	456.4500s
22	125	24	891	536.7667s
23	223	9	893	583.1167s
24	379	51	889	469.0667s
25	168	1	904	581.2833s
Average	223.440	12.120	902.920	514.992
Std. Dev.	68.483	14.492	16.466	43.729

Table C.23: Results for Benchmark c3540

c5315				
3086370 faults simulated 690 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	284	1	875	2205.0167s
2	314	28	875	2170.5167s
3	307	6	875	2262.6667s
4	444	4	875	2191.7000s
5	352	2	875	2262.5667s
6	227	1	875	2221.4667s
7	318	3	875	2236.7500s
8	366	8	875	2156.5833s
9	351	22	875	2170.9667s
10	466	1	875	2184.7333s
11	355	3	875	2098.1667s
12	394	5	875	2117.2167s
13	319	15	875	2164.1167s
14	358	1	875	2228.9167s
15	387	4	875	2089.4167s
16	293	14	875	2195.9167s
17	336	4	876	2162.7333s
18	322	1	875	2141.5667s
19	341	2	875	2133.0333s
20	299	6	875	2171.3167s
21	183	2	875	2203.1333s
22	504	8	875	2132.3000s
23	342	4	875	2209.7167s
24	395	7	875	2203.3167s
25	172	2	875	2237.3000s
Average	337.160	6.160	875.040	2182.045
Std. Dev.	74.493	6.685	0.196	46.178

Table C.24: Results for Benchmark c5315

c7552				
6913621 faults simulated 90 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	27	12	1537	1162.0833s
2	48	5	1512	980.8000s
3	48	10	1496	792.7833s
4	24	24	1608	1201.2000s
5	47	2	1497	1068.3167s
6	56	1	1503	788.0333s
7	46	28	1549	981.5333s
8	30	5	1531	1060.3667s
9	63	1	1492	928.9833s
10	37	1	1542	1219.2167s
11	77	1	1492	911.1000s
12	47	17	1514	884.4000s
13	33	1	1513	1081.6833s
14	53	1	1495	937.9333s
15	40	3	1517	819.4833s
16	38	5	1511	992.6667s
17	42	3	1516	995.4500s
18	29	4	1562	813.5833s
19	51	12	1543	817.9333s
20	43	3	1510	885.4167s
21	42	2	1508	1041.5667s
22	86	3	1490	778.5667s
23	52	4	1498	889.3833s
24	43	4	1510	897.5833s
25	51	7	1499	937.3667s
Average	46.120	6.360	1517.800	954.697
Std. Dev.	13.978	7.065	26.580	124.631

Table C.25: Results for Benchmark c7552

C.3 Single Fault Diagnosis Using Double Fault Patterns

f2				
378 faults simulated 16 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	9	0	8	0.0667s
2	9	0	11	0.0833s
3	11	0	8	0.0833s
4	13	0	8	0.0833s
5	11	0	12	0.0833s
6	15	0	8	0.0667s
7	11	0	10	0.0667s
8	11	0	8	0.0833s
9	11	0	8	0.0500s
10	11	0	8	0.0667s
Average	11.200	0.000	8.900	0.073
Std. Dev.	1.661	0.000	1.446	0.011

Table C.26: Results for Benchmark f2

rd53				
1035 faults simulated 32 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	16	0	11	0.1167s
2	24	0	10	0.1167s
3	21	0	10	0.1167s
4	21	0	10	0.1333s
5	24	0	10	0.1333s
6	24	0	10	0.1167s
7	24	0	10	0.1333s
8	24	0	10	0.1167s
9	22	0	10	0.1167s
10	24	0	10	0.1333s
Average	22.400	0.000	10.100	0.123
Std. Dev.	2.458	0.000	0.300	0.008

Table C.27: Results for Benchmark rd53

rd73				
8778 faults simulated 128 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	79	0	34	0.7167s
2	96	0	32	0.6667s
3	96	0	32	0.6000s
4	62	0	33	0.6167s
5	77	0	32	0.6667s
6	70	0	32	0.6500s
7	100	0	33	0.5833s
8	88	0	32	0.5667s
9	98	0	32	0.6500s
10	32	0	34	0.8000s
Average	79.800	0.000	32.600	0.652
Std. Dev.	20.094	0.000	0.800	0.065

Table C.28: Results for Benchmark rd73

sao2				
12090 faults simulated 1024 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	768	0	36	1.9833s
2	468	0	37	2.2167s
3	744	0	36	1.9000s
4	604	0	37	2.2667s
5	764	0	37	2.1333s
6	752	0	36	2.1833s
7	624	0	36	2.1833s
8	748	0	36	2.0167s
9	689	0	36	2.0833s
10	536	0	36	2.2000s
Average	669.700	0.000	36.300	2.117
Std. Dev.	101.151	0.000	0.458	0.112

Table C.29: Results for Benchmark sao2

bw				
12720 faults simulated 32 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	13	1	33	0.5833s
2	18	0	33	0.6667s
3	18	0	32	0.6333s
4	20	0	31	0.6333s
5	12	0	40	0.8000s
6	24	0	33	0.6167s
7	28	0	31	0.5833s
8	29	0	32	0.5833s
9	28	1	31	0.6833s
10	28	0	31	0.5167s
Average	21.800	0.200	32.700	0.630
Std. Dev.	6.145	0.400	2.571	0.073

Table C.30: Results for Benchmark bw

rd84				
25651 faults simulated 256 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	196	0	53	1.9333s
2	114	0	60	2.0833s
3	132	0	55	2.1500s
4	225	0	53	1.9667s
5	180	0	53	2.1167s
6	80	0	58	2.2833s
7	117	0	55	2.0333s
8	79	0	56	2.2167s
9	89	0	55	2.2500s
10	180	0	53	1.8333s
Average	139.200	0.000	55.100	2.087
Std. Dev.	49.745	0.000	2.256	0.139

Table C.31: Results for Benchmark rd84

9sym				
28441 faults simulated 512 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	256	0	55	2.8667s
2	163	0	55	3.3833s
3	258	0	53	3.1667s
4	160	0	57	3.6667s
5	182	0	54	3.2667s
6	285	0	53	2.9667s
7	359	0	53	2.8333s
8	384	0	53	2.6833s
9	384	0	53	2.6833s
10	383	0	53	2.9333s
Average	281.400	0.000	53.900	3.045
Std. Dev.	87.887	0.000	1.300	0.304

Table C.32: Results for Benchmark 9sym

c432				
19110 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	749	0	40	6.3833s
2	754	0	40	6.5000s
3	742	0	40	5.8667s
4	417	0	40	6.0833s
5	662	0	41	6.4333s
6	645	0	40	6.5333s
7	644	0	40	6.0833s
8	632	0	40	5.9833s
9	756	0	40	5.8500s
10	727	0	40	5.9167s
Average	672.800	0.000	40.100	6.163
Std. Dev.	97.911	0.000	0.300	0.258

Table C.33: Results for Benchmark c432

c499				
29403 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	729	0	40	8.6833s
2	751	0	40	9.1833s
3	740	0	40	8.3167s
4	752	0	40	8.6333s
5	756	0	40	8.3000s
6	733	0	40	9.5500s
7	756	0	40	8.2167s
8	731	0	40	8.0500s
9	743	0	40	8.3000s
10	748	0	40	8.2167s
Average	743.900	0.000	40.000	8.545
Std. Dev.	9.741	0.000	0.000	0.456

Table C.34: Results for Benchmark c499

c880				
97903 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	506	0	82	18.6167s
2	655	0	81	17.5833s
3	569	0	84	17.8667s
4	681	0	81	18.4667s
5	757	0	84	19.1833s
6	512	0	83	19.6333s
7	720	0	84	16.7333s
8	632	0	82	18.1333s
9	742	0	81	16.9167s
10	724	0	82	17.9667s
Average	649.800	0.000	82.400	18.110
Std. Dev.	88.283	0.000	1.200	0.868

Table C.35: Results for Benchmark c880

c1355				
171991 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	615	0	68	65.8667s
2	671	0	74	50.2833s
3	732	0	64	49.5167s
4	437	0	78	84.2667s
5	737	0	62	50.0667s
6	500	0	82	79.5833s
7	619	0	76	85.1167s
8	437	0	85	57.7167s
9	604	0	72	67.7000s
10	690	0	62	67.6667s
Average	604.200	0.000	72.300	65.778
Std. Dev.	106.253	0.000	7.772	13.153

Table C.36: Results for Benchmark c1355

c1908				
416328 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	741	0	550	211.6833s
2	755	0	552	198.4333s
3	745	0	542	191.9167s
4	740	0	548	201.7833s
5	748	0	544	184.6833s
6	746	0	550	214.1667s
7	745	0	549	193.2167s
8	729	0	548	212.2667s
9	744	0	545	217.0167s
10	738	0	555	213.4333s
Average	743.100	0.000	548.300	203.860
Std. Dev.	6.488	0.000	3.662	10.772

Table C.37: Results for Benchmark c1908

c2670				
1016025 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	543	0	573	442.9833s
2	742	0	573	438.4667s
3	746	0	573	456.8167s
4	727	0	573	441.5667s
5	571	0	573	466.0000s
6	751	0	573	459.3833s
7	722	0	573	445.7667s
8	736	0	573	453.1000s
9	742	0	573	441.3667s
10	558	0	573	467.7500s
Average	683.800	0.000	573.000	451.320
Std. Dev.	83.417	0.000	0.000	10.216

Table C.38: Results for Benchmark c2670

c3540				
1476621 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	540	0	853	951.8833s
2	777	0	861	912.4167s
3	682	0	849	936.4333s
4	580	0	861	998.4333s
5	676	0	855	929.8333s
6	751	0	849	988.3667s
7	778	0	853	990.5500s
8	557	0	872	995.1333s
9	672	0	853	1012.9000s
10	767	0	849	924.8833s
Average	678.000	0.000	855.500	964.083
Std. Dev.	87.405	0.000	6.917	34.799

Table C.39: Results for Benchmark c3540

c5315				
3086370 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	688	0	875	3184.9000s
2	735	0	875	3199.1833s
3	657	0	875	3265.3167s
4	748	0	875	3135.5500s
5	733	0	875	3169.8167s
6	711	0	875	3171.5333s
7	742	0	875	3202.7333s
8	614	0	875	3207.4333s
9	637	0	875	3234.6667s
10	712	0	875	3198.2667s
Average	697.700	0.000	875.000	3196.940
Std. Dev.	44.668	0.000	0.000	33.976

Table C.40: Results for Benchmark c5315

c7552				
6913621 faults simulated 1000 test vectors				
Pattern	Frequency	F_L	F_S	CPU Time
1	692	0	1477	9006.5333s
2	697	0	1481	9123.2833s
3	654	0	1475	9221.7333s
4	689	0	1475	8966.4667s
5	734	0	1477	8966.4667s
6	693	0	1475	8923.1833s
7	747	0	1475	8982.0333s
8	719	0	1475	9079.7667s
9	551	0	1477	9283.0167s
10	721	0	1475	8958.1167s
Average	689.700	0.000	1476.200	9051.060
Std. Dev.	52.599	0.000	1.833	116.200

Table C.41: Results for Benchmark c7552

Appendix D

Multiple Fault Algorithm Simulation Tables

D.1 Multiple Fault Diagnosis Using Single Fault Patterns

f2								
378 faults simulated 16 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	6	2	8	10	0	2	8	0.0833s
2	6	2	7	8	0	0	3	0.1000s
3	10	1	13	26	0	89	4	0.1000s
4	12	2	29	36	0	283	18	0.1667s
5	10	1	10	20	0	81	3	0.0833s
6	12	2	125	18	0	177	126	0.5833s
7	4	2	0	10	0	1	0	0.0667s
8	8	1	2	27	0	76	1	0.0667s
9	8	1	10	11	0	7	4	0.0667s
10	12	1	29	34	0	291	23	0.2000s
11	8	1	15	8	0	0	4	0.0833s
12	6	2	0	9	0	0	0	0.0500s
13	14	2	103	41	0	466	103	0.6000s
14	8	1	18	23	0	148	12	0.1167s
15	8	1	1	22	0	2	0	0.0833s
16	6	4	4	8	0	0	5	0.0833s
17	10	2	13	27	0	203	16	0.1333s
18	4	1	0	8	0	0	0	0.0667s
19	10	2	14	18	0	52	4	0.0833s
20	8	1	4	21	0	28	2	0.0833s
21	8	1	14	10	0	19	11	0.0833s
22	12	4	58	24	0	255	122	0.2500s
23	8	1	19	11	0	33	33	0.0833s
24	10	2	15	21	0	99	9	0.1167s
25	4	1	0	8	0	0	0	0.0833s
26	8	1	7	18	0	55	2	0.0833s
27	8	1	7	26	0	69	2	0.0667s
28	10	2	13	27	0	203	16	0.1000s
29	4	4	0	8	0	0	0	0.0833s
30	10	1	21	21	0	109	14	0.1333s
Average	8.400	1.667	18.633	18.633	0.000	91.600	18.167	0.133
Std. Dev.	2.603	0.907	28.158	9.214	0.000	113.374	33.984	0.129

Table D.1: Results for Benchmark f2

rd53								
1035 faults simulated 32 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	20	1	14	62	0	296	4	0.2833s
2	8	2	3	54	0	121	2	0.2000s
3	12	2	3	50	0	151	1	0.2000s
4	16	1	4	48	0	151	2	0.2167s
5	24	2	73	88	0	1107	201	0.8167s
6	16	1	1	48	0	154	0	0.2000s
7	16	1	3	45	0	75	0	0.1833s
8	8	2	0	45	0	130	0	0.2500s
9	16	2	17	45	0	97	6	0.2333s
10	5	1	0	45	0	87	0	0.2000s
11	16	1	0	51	0	140	0	0.1833s
12	16	2	4	48	0	161	2	0.1833s
13	18	1	26	63	0	430	54	0.3833s
14	16	1	0	45	0	98	0	0.1833s
15	16	1	1	45	0	74	0	0.1667s
16	16	1	3	51	0	170	0	0.2500s
17	19	1	17	62	0	378	10	0.2833s
18	16	2	7	50	0	98	4	0.2333s
19	16	2	0	48	0	93	0	0.1833s
20	16	2	2	49	0	97	0	0.1667s
21	10	1	0	47	0	89	0	0.1833s
22	16	1	0	52	0	187	0	0.2000s
23	16	1	8	48	0	195	12	0.2167s
24	16	1	0	50	0	110	0	0.2000s
25	24	2	74	66	0	455	84	0.8500s
26	24	1	37	61	0	307	14	0.3667s
27	20	1	8	56	0	237	0	0.2333s
28	20	1	8	50	0	145	1	0.2000s
29	16	1	6	49	0	111	0	0.2667s
30	16	1	6	51	0	189	1	0.2667s
Average	16.133	1.333	10.833	52.400	0.000	204.433	13.267	0.266
Std. Dev.	4.295	0.471	18.700	8.842	0.000	195.551	38.994	0.160

Table D.2: Results for Benchmark rd53

rd73								
8778 faults simulated 128 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	64	1	2	104	0	251	0	1.0833s
2	92	1	57	144	0	739	0	1.8500s
3	66	1	6	107	0	369	0	1.2000s
4	80	1	8	110	0	230	0	1.1333s
5	82	1	20	106	0	212	0	1.2500s
6	78	1	5	104	0	295	0	1.1500s
7	70	1	111	125	0	1282	27	2.3833s
8	56	1	1	103	0	354	0	1.1833s
9	104	1	190	162	0	1022	5	4.4167s
10	64	3	18	105	0	310	16	1.3000s
11	72	1	3	107	0	253	0	1.1333s
12	64	1	2	101	0	233	0	1.1000s
13	46	1	23	114	0	447	11	1.3667s
14	24	1	10	108	0	517	7	1.3667s
15	65	1	9	107	0	238	0	1.1667s
16	38	1	1	103	0	264	0	1.1833s
17	48	1	0	99	0	224	0	1.1167s
18	36	1	8	107	0	368	0	1.3000s
19	96	1	80	133	0	771	1	2.1500s
20	64	1	6	106	0	237	1	1.1333s
21	64	2	22	117	0	282	4	1.3667s
22	64	2	15	107	0	355	6	1.2000s
23	88	2	39	119	0	378	0	1.4333s
24	106	1	249	139	0	1140	107	5.5167s
25	48	2	8	114	0	503	0	1.4167s
26	112	1	397	175	0	1642	11	10.6333s
27	64	1	34	101	0	364	12	1.3333s
28	64	1	6	112	0	323	0	1.1833s
29	2	1	0	98	0	66	0	1.2167s
30	30	2	12	103	0	408	11	1.3000s
Average	65.033	1.233	44.733	114.667	0.000	469.233	7.300	1.886
Std. Dev.	24.400	0.496	86.208	18.207	0.000	355.292	19.554	1.887

Table D.3: Results for Benchmark rd73

sao2								
12090 faults simulated 1024 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	512	1	38	75	0	377	32	3.5167s
2	976	2	25	95	0	515	0	2.2833s
3	512	1	1	64	0	80	0	2.8667s
4	1004	1	175	190	0	1630	0	8.5167s
5	192	2	0	66	0	110	0	3.6667s
6	352	1	9	64	0	55	0	3.4500s
7	512	1	51	73	0	224	24	3.6167s
8	480	1	0	64	0	79	0	2.8833s
9	696	1	175	69	0	227	3	5.7333s
10	960	1	172	119	0	613	1	5.5833s
11	544	1	12	73	0	208	3	3.1500s
12	120	1	2	66	0	132	0	3.9167s
13	720	1	113	81	0	295	14	4.2500s
14	800	1	166	79	0	284	0	5.3333s
15	320	1	5	64	0	63	4	3.2833s
16	832	1	293	149	0	1809	327	13.2333s
17	508	2	31	81	0	180	12	3.5667s
18	480	1	2	64	0	81	0	3.0000s
19	512	1	0	64	0	121	0	2.8167s
20	256	1	1	64	0	82	0	3.5667s
21	560	1	145	86	0	443	32	5.3000s
22	400	1	11	66	0	53	0	3.2000s
23	512	1	22	64	0	81	0	3.1167s
24	508	1	26	66	0	80	2	3.2833s
25	392	1	0	64	0	76	0	3.4167s
26	457	1	0	64	0	123	0	2.8333s
27	76	1	2	64	0	90	1	3.8167s
28	680	1	109	72	0	200	0	3.8833s
29	498	2	0	68	0	147	0	2.9833s
30	566	1	1	66	0	129	0	2.9333s
Average	531.233	1.133	52.900	78.133	0.000	286.233	15.167	4.100
Std. Dev.	227.325	0.340	75.969	27.562	0.000	408.086	58.604	2.086

Table D.4: Results for Benchmark sao2

bw								
12720 faults simulated 32 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	6	2	10	94	0	679	3	1.1333s
2	20	1	178	111	0	1875	168	3.0000s
3	10	1	45	87	0	626	38	1.2667s
4	13	2	158	111	0	1905	265	2.7167s
5	24	1	400	217	0	6764	733	13.8333s
6	10	1	16	85	0	415	0	1.1833s
7	8	1	20	83	0	468	0	1.1167s
8	20	2	97	120	0	1698	1	2.0333s
9	12	1	181	109	0	2531	117	3.1500s
10	20	1	117	109	0	1261	0	2.0833s
11	9	1	10	89	0	445	1	1.1500s
12	18	1	121	123	0	1985	9	2.4833s
13	24	1	370	214	0	4593	63	10.8500s
14	24	1	624	173	0	6706	969	20.2167s
15	18	1	227	155	0	4246	246	4.4167s
16	22	1	811	169	0	12336	1402	31.9333s
17	12	1	30	87	0	708	3	1.2667s
18	22	1	120	150	0	2309	0	2.7833s
19	25	1	584	221	0	8114	139	18.7667s
20	20	1	221	118	0	2351	1	3.5167s
21	22	1	84	113	0	1152	20	1.8833s
22	12	1	63	98	0	909	41	1.4833s
23	23	1	230	149	0	3240	69	4.5167s
24	20	1	805	123	0	6260	2007	35.6167s
25	22	1	241	134	0	2790	319	4.8333s
26	2	1	0	79	0	140	0	1.0667s
27	24	2	548	243	0	13972	1658	18.0000s
28	11	1	88	96	0	872	172	1.6500s
29	24	1	603	252	0	18683	3835	23.0500s
30	24	1	312	159	0	2511	134	8.8167s
Average	17.367	1.133	243.800	135.700	0.000	3751.467	413.767	7.661
Std. Dev.	6.463	0.340	237.110	49.370	0.000	4364.251	819.423	9.440

Table D.5: Results for Benchmark bw

rd84								
25651 faults simulated 256 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	136	2	4	174	1	820	0	3.5833s
2	128	1	4	179	1	1264	0	3.8667s
3	82	1	5	185	1	1161	1	4.4833s
4	37	1	1	179	1	1423	0	4.3667s
5	176	1	72	201	1	1495	51	4.9667s
6	40	1	12	177	1	1182	8	4.4167s
7	208	1	1551	282	1	6521	0	101.4833s
8	180	1	109	235	1	1880	0	6.3000s
9	96	1	30	178	1	1371	14	4.4333s
10	238	1	2659	428	1	21813	2992	442.3000s
11	156	1	5	195	1	1472	0	3.5667s
12	193	1	16	195	1	1055	0	3.4833s
13	160	1	12	191	1	1162	0	3.7667s
14	186	1	474	296	1	4020	12	20.0667s
15	120	1	19	200	1	1319	7	4.5000s
16	206	1	264	252	1	2372	1	10.6833s
17	128	1	29	199	1	1127	0	4.0500s
18	37	1	20	181	1	1234	13	4.5833s
19	252	1	8074	571	1	58069	8296	2226.8500s
20	48	1	0	183	1	906	0	4.6333s
21	68	1	7	191	1	1127	0	4.2667s
22	74	1	4	181	1	1093	0	4.0167s
23	56	2	37	185	1	738	8	4.8500s
24	28	2	0	187	1	1226	0	4.4000s
25	62	1	0	186	1	1409	0	4.3167s
26	28	1	1	199	1	1245	0	4.7500s
27	104	1	21	180	1	1040	2	4.1333s
28	179	1	24	178	1	1222	9	3.4333s
29	180	1	106	236	1	2252	12	6.4167s
30	128	1	15	193	1	1401	0	4.0667s
Average	123.800	1.100	452.500	219.900	1.000	4147.300	380.867	97.034
Std. Dev.	65.764	0.300	1514.938	82.554	0.000	10705.844	1564.475	403.457

Table D.6: Results for Benchmark rd84

9sym								
28441 faults simulated 512 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	256	1	10	130	0	894	0	4.0833s
2	130	1	0	121	0	712	0	4.4167s
3	160	1	3	124	0	747	0	4.3500s
4	256	1	0	123	0	730	0	3.9500s
5	122	1	0	123	0	720	0	4.4667s
6	48	2	1	120	0	808	0	4.7833s
7	288	1	8	127	0	820	3	4.1333s
8	256	1	4	128	0	711	0	3.9667s
9	320	1	37	141	0	1129	3	4.2667s
10	256	1	3	134	0	1215	1	4.0833s
11	352	1	46	140	0	999	2	4.5833s
12	256	1	14	133	0	868	0	4.4167s
13	256	1	2	128	0	909	0	4.0833s
14	406	2	28	151	0	1118	0	4.1333s
15	365	1	83	148	0	1085	4	4.9333s
16	188	1	5	126	0	842	2	4.3667s
17	256	1	0	124	0	719	0	3.9667s
18	88	1	1	123	0	738	0	4.5167s
19	280	1	48	145	0	1306	27	4.9000s
20	76	1	2	123	0	759	1	4.8500s
21	92	1	0	120	0	658	0	4.6167s
22	56	1	0	118	0	468	0	4.6833s
23	160	1	5	128	0	1058	1	4.6000s
24	52	2	2	125	0	899	1	5.0833s
25	352	2	32	146	0	947	10	4.2167s
26	448	2	447	204	0	2811	226	19.5833s
27	256	2	9	123	0	729	3	4.1000s
28	220	2	4	123	0	473	0	4.4833s
29	256	1	23	136	0	1116	4	4.5167s
30	389	1	8	129	0	745	0	3.7167s
Average	229.867	1.233	27.500	132.133	0.000	924.433	9.600	4.895
Std. Dev.	110.024	0.423	80.157	16.043	0.000	402.107	40.502	2.747

Table D.7: Results for Benchmark 9sym

c432								
19110 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	507	1	0	103	8	180	0	6.2833s
2	495	1	0	103	8	282	0	6.2167s
3	501	1	0	103	8	180	0	6.2333s
4	517	2	0	103	8	180	0	6.2000s
5	537	1	0	103	8	180	0	6.0667s
6	477	1	0	103	8	180	0	6.4333s
7	514	1	0	103	8	180	0	6.0833s
8	219	1	0	103	8	270	0	7.3000s
9	254	2	0	103	8	180	0	7.3333s
10	495	1	0	103	8	180	0	6.8667s
11	316	3	49	157	8	496	24	10.6667s
12	532	2	0	103	8	180	0	6.1833s
13	820	1	12	97	8	183	1	4.5333s
14	482	2	0	103	8	181	0	6.2333s
15	694	1	2	102	8	180	0	5.1833s
16	677	1	3	126	8	386	0	5.7167s
17	543	1	0	108	8	220	0	6.1333s
18	528	1	0	103	8	190	0	6.0667s
19	544	1	0	107	8	224	0	6.5167s
20	344	1	0	103	8	180	0	6.9167s
21	501	1	0	103	8	180	0	6.2167s
22	234	1	0	103	8	277	0	7.3833s
23	508	1	2	103	8	201	1	6.0000s
24	557	1	0	103	8	194	0	5.9667s
25	241	1	0	106	8	281	0	7.4667s
26	499	2	0	103	8	181	0	6.0833s
27	771	1	1	103	8	180	0	4.9167s
28	524	2	0	103	8	180	0	6.2167s
29	441	1	0	106	8	209	0	6.3833s
30	502	1	0	103	8	180	0	6.0667s
Average	492.467	1.267	2.300	105.833	8.000	215.833	0.867	6.396
Std. Dev.	140.758	0.512	8.952	10.488	0.000	70.002	4.303	1.022

Table D.8: Results for Benchmark c432

c499								
29403 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	498	1	9	1653	81	1703	0	165.9667s
2	496	1	14	1653	81	1259	0	167.0167s
3	496	2	14	1651	81	992	0	167.4667s
4	510	5	14	1646	81	1000	0	165.0500s
5	498	1	9	1654	81	1973	0	163.4667s
6	491	1	44	1647	81	964	0	166.2500s
7	492	2	9	1652	81	964	0	166.7667s
8	518	5	32	1644	81	2475	35	163.9333s
9	487	1	9	1655	81	1584	0	167.0167s
10	496	1	35	1655	81	2376	0	171.3167s
11	509	1	35	1648	81	962	0	164.1667s
12	483	5	505	1639	81	1065	1565	232.0333s
13	501	1	20	1650	81	968	0	164.5333s
14	480	1	9	1655	81	1957	0	167.6333s
15	480	1	14	1651	81	1385	0	163.8333s
16	506	1	5	1653	81	960	0	160.7833s
17	482	5	213	1638	81	980	0	193.3333s
18	498	1	90	1647	81	1035	1	171.9000s
19	501	5	475	1639	81	1015	345	232.5833s
20	494	5	14	1651	81	970	0	166.8333s
21	499	5	20	1651	81	2155	25	166.9000s
22	490	1	20	1651	81	1382	0	163.6000s
23	503	1	9	1652	81	961	0	164.8667s
24	508	1	9	1655	81	1094	0	164.7833s
25	512	1	32	1649	81	3045	32	166.7333s
26	502	26	156	1639	81	1064	0	185.0000s
27	486	5	27	1649	81	970	0	164.8833s
28	512	5	14	1646	81	1025	0	164.2500s
29	509	1	27	1649	81	989	0	170.6333s
30	511	1	27	1649	81	960	0	170.1000s
Average	498.267	3.100	63.667	1649.033	81.000	1341.067	66.767	172.121
Std. Dev.	10.230	4.614	122.262	4.929	0.000	553.345	285.038	17.273

Table D.9: Results for Benchmark c499

c880								
97903 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	502	1	11	445	75	1318	0	27.9500s
2	332	1	9	430	75	1251	0	30.5833s
3	709	4	13	441	75	1669	4	24.4667s
4	252	2	26	450	75	3921	29	33.6500s
5	306	1	3	447	75	1634	0	31.9167s
6	485	1	8	438	75	1516	0	28.0167s
7	145	2	13	428	75	686	0	34.1500s
8	497	1	27	460	75	1646	0	29.3667s
9	512	1	8	425	75	856	0	27.2833s
10	555	1	2	438	75	1217	0	27.2333s
11	497	1	23	437	75	1552	0	28.2000s
12	646	2	43	476	75	1536	6	27.6833s
13	375	1	11	447	75	1453	0	30.7667s
14	694	4	25	463	75	1648	0	25.4167s
15	609	1	18	453	75	1762	0	26.7833s
16	497	1	31	454	75	1396	0	29.2500s
17	482	1	8	425	75	455	0	27.8667s
18	549	1	21	430	75	1176	0	27.1833s
19	66	4	9	424	75	1579	4	35.5833s
20	484	1	3	434	75	942	0	27.8167s
21	489	1	5	442	75	1549	0	28.4667s
22	501	2	21	444	75	1540	0	28.7167s
23	436	1	3	436	75	1294	0	29.1667s
24	471	2	10	445	75	1474	0	28.6167s
25	302	1	43	466	75	2318	2	34.2667s
26	498	1	4	439	75	1810	0	28.0167s
27	487	1	13	445	75	1835	0	29.5833s
28	464	2	14	457	75	1469	0	29.7167s
29	477	2	23	451	75	1551	0	29.1333s
30	481	2	4	423	75	545	0	27.3333s
Average	460.000	1.567	15.067	443.100	75.000	1486.600	1.500	29.139
Std. Dev.	138.674	0.920	11.000	13.027	0.000	595.506	5.309	2.536

Table D.10: Results for Benchmark c880

c1355								
171991 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	251	1	50	2977	122	90395	0	638.9000s
2	493	1	391	3020	122	145202	208	664.2167s
3	508	2	97	2965	122	115393	0	557.4667s
4	371	1	68	2968	122	95829	0	588.8667s
5	509	1	170	2959	122	103970	0	568.1833s
6	480	1	313	3068	122	111967	6	641.4000s
7	501	2	189	2966	122	101609	0	596.6167s
8	515	2	275	2959	122	130417	0	623.6667s
9	490	2	145	2965	122	111473	0	653.5333s
10	507	5	241	2973	122	97356	0	637.8833s
11	124	1	44	2993	122	70088	49	655.0167s
12	355	1	83	2965	122	91504	0	606.8833s
13	505	2	263	2963	122	124218	0	592.6667s
14	499	2	294	3020	122	127826	30	634.2167s
15	476	1	86	2969	122	115300	0	548.0833s
16	499	1	254	3116	122	136759	307	631.5667s
17	292	1	18	2968	122	85916	0	607.1333s
18	5	1	10	2968	122	687953	25	671.8667s
19	249	1	47	2966	122	87204	0	646.9333s
20	504	5	286	2973	122	83298	0	623.7000s
21	514	2	227	2950	122	126054	0	587.0167s
22	251	1	77	3039	122	108022	116	650.0167s
23	384	1	115	2983	122	111809	0	618.6833s
24	239	1	57	2997	122	148303	55	650.3167s
25	481	5	215	2971	122	76037	0	603.5000s
26	518	1	265	2953	122	122368	56	600.6000s
27	382	1	86	2962	122	118178	0	608.9667s
28	751	1	595	3061	122	121956	0	670.3000s
29	671	1	416	2953	122	128748	0	587.9667s
30	492	1	135	2959	122	108118	0	583.9500s
Average	427.200	1.633	183.733	2984.967	122.000	129442.333	28.400	618.337
Std. Dev.	151.175	1.197	133.897	38.863	0.000	105490.548	67.551	32.681

Table D.11: Results for Benchmark c1355

c1908								
416328 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	502	5	322	5231	1085	70663	0	1733.3833s
2	484	8	260	5269	1085	78898	108	1955.8667s
3	507	2	276	5215	1085	74123	0	1678.2667s
4	514	24	457	5250	1085	92846	194	1791.5333s
5	515	8	335	5216	1085	84857	0	1733.8833s
6	499	8	364	5207	1085	81897	0	1736.2667s
7	515	3	484	5239	1085	102658	14	1808.6333s
8	521	1	277	5199	1085	74652	0	1891.3333s
9	526	2	534	5250	1085	137140	1344	1843.9333s
10	480	15	216	5207	1085	94256	0	1699.2667s
11	509	6	521	5269	1085	92630	0	1854.7833s
12	524	22	331	5199	1085	69580	0	1971.7333s
13	491	6	460	5268	1085	92769	0	2114.7833s
14	506	119	847	5332	1085	116316	1762	2112.2833s
15	504	12	291	5196	1085	77145	0	1694.8167s
16	516	26	572	5316	1085	118236	272	1904.6000s
17	497	21	545	5231	1085	89670	249	2067.6833s
18	479	6	702	5325	1085	98368	0	2274.6500s
19	491	18	457	5204	1085	70263	0	2045.9667s
20	446	5	403	5242	1085	84589	0	1806.6333s
21	507	6	262	5207	1085	79734	0	1942.8667s
22	477	25	369	5343	1085	131732	625	2093.9833s
23	483	39	560	5241	1085	85787	0	2141.0000s
24	502	6	832	5210	1085	87856	0	2245.1333s
25	736	1	793	5279	1085	94605	10	2091.1333s
26	483	5	902	5216	1085	95994	20	2168.5667s
27	511	33	240	5281	1085	93864	0	2006.1833s
28	490	2	30	5197	1085	79356	0	1604.0000s
29	481	2	192	5194	1085	74756	0	1878.0000s
30	521	12	697	5326	1085	100623	18	2244.0167s
Average	507.233	14.933	451.033	5245.300	1085.000	90862.100	153.867	1937.839
Std. Dev.	45.926	21.752	212.768	45.154	0.000	16771.706	399.068	185.937

Table D.12: Results for Benchmark c1908

c2670								
1016025 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	463	6	10	2760	954	12951	24	651.2167s
2	163	4	398	2810	954	23046	4840	864.4167s
3	489	14	0	2746	954	13034	0	659.0333s
4	541	3	0	2754	954	12581	0	668.8667s
5	662	1	12	2766	954	12401	0	570.1500s
6	494	1	0	2741	954	12357	0	645.2833s
7	480	2	0	2751	954	20615	0	641.1167s
8	529	6	0	2763	954	12385	0	619.7667s
9	548	8	0	2754	954	12347	0	613.9500s
10	495	18	12	2755	954	12527	0	629.9333s
11	472	2	0	2769	954	12664	0	647.7167s
12	516	6	12	2751	954	12420	0	632.2500s
13	493	2	24	2759	954	12292	0	628.6667s
14	239	2	0	2752	954	14609	0	743.7833s
15	432	4	32	2859	954	23730	64	689.5000s
16	564	5	216	2807	954	13179	5	666.0167s
17	470	1	0	2742	954	12537	0	636.3167s
18	503	1	0	2760	954	12373	0	635.7333s
19	489	2	0	2744	954	12399	0	626.3500s
20	473	2	0	2769	954	12382	0	649.5167s
21	456	20	0	2783	954	13110	0	649.3500s
22	816	2	274	3074	954	16161	0	632.7333s
23	514	6	0	2757	954	12398	0	625.1833s
24	486	2	0	2755	954	12473	0	657.2833s
25	482	7	0	2744	954	12709	0	738.9167s
26	176	4	0	2734	954	17300	0	774.3500s
27	476	21	14	2769	954	13242	0	645.5167s
28	780	24	264	3040	954	34363	3316	651.3833s
29	534	2	0	2767	954	12413	0	620.0333s
30	523	12	0	2777	954	12456	0	631.0333s
Average	491.933	6.333	42.267	2783.733	954.000	14648.467	274.967	658.179
Std. Dev.	130.700	6.472	99.761	77.069	0.000	4780.926	1035.341	55.213

Table D.13: Results for Benchmark c2670

c3540								
1476621 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	467	8	59	3883	2201	87887	0	1371.0000s
2	126	1	0	3830	2201	73854	0	1660.8833s
3	507	1	30	3911	2201	82567	0	1339.3500s
4	568	9	140	3978	2201	100281	0	1349.8167s
5	513	16	26	3875	2201	95718	0	1362.4667s
6	347	2	67	3899	2201	95890	0	1539.7500s
7	715	16	486	4052	2201	116752	0	1366.8667s
8	528	2	59	3905	2201	103819	0	1324.5167s
9	281	8	215	3863	2201	47260	194	1584.1500s
10	396	1	65	3920	2201	88500	0	1460.0833s
11	353	4	7	3844	2201	80517	0	1420.8667s
12	516	1	46	3872	2201	85731	0	1315.8833s
13	506	18	57	3901	2201	87129	0	1321.1500s
14	534	4	26	3923	2201	89315	0	1296.0833s
15	518	2	25	3835	2201	96871	0	1267.5333s
16	633	16	70	3881	2201	103055	0	1196.7000s
17	512	1	90	3909	2201	93198	0	1338.3833s
18	558	24	19	3878	2201	75973	0	1270.8000s
19	612	19	1	3848	2201	73948	0	1212.1333s
20	450	28	42	3880	2201	42762	0	1356.5000s
21	742	1	222	3990	2201	89718	0	1151.7000s
22	243	24	9	3855	2201	103449	0	1582.3500s
23	513	9	0	3831	2201	75795	0	1292.1333s
24	825	51	90	3940	2201	87244	0	1055.0833s
25	335	1	37	3857	2201	104496	0	1451.9667s
26	764	19	314	3992	2201	108385	0	1180.5667s
27	823	9	167	3950	2201	106816	309	1073.1333s
28	527	2	14	3859	2201	89773	0	1283.7167s
29	523	6	33	3884	2201	91568	0	1305.6167s
30	793	2	168	3989	2201	87230	0	1104.0000s
Average	524.267	10.167	86.133	3901.133	2201.000	88850.033	16.767	1327.839
Std. Dev.	166.905	11.130	105.416	54.388	0.000	15760.114	64.468	143.581

Table D.14: Results for Benchmark c3540

c5315								
3086370 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	423	1	1	6289	1461	40651	0	3600.6833s
2	484	28	0	6284	1461	38805	0	3445.0333s
3	463	6	2	6297	1461	39018	0	3582.1667s
4	650	4	19	6325	1461	39290	0	3195.0833s
5	502	2	5	6289	1461	40295	0	3574.4833s
6	357	1	13	6281	1461	40248	0	3995.3667s
7	475	3	1	6296	1461	39179	0	3550.1000s
8	525	8	3	6297	1461	41528	0	3362.8833s
9	514	22	1	6284	1461	39342	0	3663.2000s
10	685	1	7	6293	1461	38843	0	2981.1667s
11	511	3	0	6286	1461	39317	0	3325.8500s
12	544	5	5	6323	1461	39529	0	3304.2333s
13	463	15	0	6298	1461	39611	0	3496.9500s
14	512	1	1	6303	1461	39132	0	3442.4167s
15	539	4	1	6305	1461	39112	0	3335.5000s
16	447	14	9	6294	1461	38956	0	3798.5167s
17	484	4	1	6285	1461	39351	0	3464.0000s
18	485	1	5	6284	1461	40365	0	3434.6333s
19	491	2	0	6282	1461	38989	0	3573.5333s
20	462	6	2	6286	1461	39171	0	3478.7833s
21	284	2	0	6285	1461	43442	0	4111.4333s
22	724	8	5	6307	1461	40262	0	2708.4667s
23	480	4	3	6295	1461	40476	0	3490.6167s
24	545	7	5	6293	1461	39510	0	3359.7833s
25	275	2	9	6281	1461	40389	0	4274.1000s
26	450	12	3	6281	1461	40984	0	3536.2333s
27	475	6	1	6286	1461	38934	0	3776.2667s
28	585	1	7	6291	1461	38990	0	3537.5167s
29	528	2	13	6310	1461	39393	0	4093.0833s
30	661	2	19	6305	1461	38854	0	3296.3500s
Average	500.767	5.900	4.700	6293.833	1461.000	39732.200	0.000	3526.281
Std. Dev.	97.126	6.353	5.242	11.481	0.000	986.129	0.000	313.657

Table D.15: Results for Benchmark c5315

c7552								
6913621 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	414	12	24	12531	3199	195146	0	14065.1167s
2	487	5	49	12524	3199	176932	0	13046.7000s
3	521	10	80	12524	3199	180456	0	12143.4167s
4	358	20	270	14844	3199	757455	1142	17457.3167s
5	514	2	103	12541	3199	175865	0	12510.5333s
6	548	1	161	12551	3199	191314	0	12154.9167s
7	506	28	59	12549	3199	201749	52	12713.6833s
8	451	5	297	12546	3199	200711	30	13429.4833s
9	592	1	196	12587	3199	189016	0	12133.9167s
10	368	1	134	12509	3199	182011	0	13674.7333s
11	818	1	18937	19406	3199	1192790	7650	61292.6333s
12	510	17	50	12565	3199	178144	0	12663.2833s
13	350	1	190	12524	3199	200896	0	14315.4000s
14	504	1	194	12533	3199	183179	0	12604.5833s
15	474	3	24	12521	3199	181604	0	12203.1833s
16	465	5	132	12534	3199	193043	0	13175.3333s
17	511	3	50	12532	3199	181252	0	14868.0500s
18	427	4	132	12556	3199	199796	0	12981.2500s
19	514	12	193	12542	3199	182879	0	12406.3833s
20	497	3	108	12529	3199	189827	0	12374.3333s
21	494	2	227	12534	3199	190085	0	12274.1667s
22	910	3	1643	12937	3199	200091	1113	10793.6833s
23	475	4	192	12545	3199	199679	0	12738.2333s
24	463	4	53	12526	3199	182371	0	12770.2667s
25	523	7	50	12529	3199	178417	0	12315.4000s
26	161	2	55	12531	3199	800339	0	15212.7667s
27	449	27	82	12553	3199	208035	0	13290.5667s
28	561	51	221	12537	3199	197216	0	12220.2667s
29	506	1	222	12548	3199	189545	0	12709.2667s
30	423	2	24	12511	3199	180803	0	13185.9167s
Average	493.133	7.933	805.067	12856.633	3199.000	262021.533	332.900	14657.493
Std. Dev.	127.377	10.853	3378.844	1285.919	0.000	227065.324	1387.376	8741.692

Table D.16: Results for Benchmark c7552

D.2 Multiple Fault Diagnosis Using Double Fault Patterns

f2								
378 faults simulated 16 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	9	0	26	11	0	34	28	0.1167s
2	9	0	11	29	0	144	10	0.1000s
3	11	0	36	11	0	29	2	0.1167s
4	13	0	71	38	0	524	212	0.3667s
5	11	0	22	25	0	133	7	0.1333s
6	15	0	183	42	0	734	648	1.2500s
7	11	0	14	28	0	126	10	0.1167s
8	11	0	106	14	0	124	530	0.4167s
9	11	0	15	19	0	97	2	0.0667s
10	11	0	29	37	0	348	14	0.1667s
Average	11.200	0.000	51.300	25.400	0.000	229.300	146.300	0.285
Std. Dev.	1.661	0.000	52.238	10.837	0.000	221.299	230.857	0.340

Table D.17: Results for Benchmark f2

rd53								
1035 faults simulated 32 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	16	0	8	61	0	334	8	0.2833s
2	24	0	33	62	0	260	1	0.3833s
3	21	0	12	46	0	75	2	0.2167s
4	21	0	27	47	0	132	2	0.2833s
5	24	0	17	61	0	239	2	0.3000s
6	24	0	5	48	0	69	1	0.2000s
7	24	0	31	59	0	236	3	0.3667s
8	24	0	11	53	0	190	8	0.2333s
9	22	0	23	58	0	254	6	0.2833s
10	24	0	21	55	0	187	6	0.2667s
Average	22.400	0.000	18.800	55.000	0.000	197.600	3.900	0.282
Std. Dev.	2.458	0.000	9.261	5.865	0.000	80.567	2.663	0.056

Table D.18: Results for Benchmark rd53

rd73								
8778 faults simulated 128 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	79	0	54	112	0	337	1	1.4333s
2	96	0	128	138	0	874	1	2.9167s
3	96	0	42	108	0	228	7	1.3667s
4	62	0	59	118	0	540	1	1.7167s
5	77	0	52	112	0	312	1	1.4667s
6	70	0	11	110	0	451	1	1.2833s
7	100	0	57	122	0	324	2	1.7167s
8	88	0	39	126	0	449	4	1.5500s
9	98	0	234	147	0	903	4	5.1667s
10	32	0	16	103	0	425	4	1.3333s
Average	79.800	0.000	69.200	119.600	0.000	484.300	2.600	1.995
Std. Dev.	20.094	0.000	62.662	13.253	0.000	218.842	1.960	1.149

Table D.19: Results for Benchmark rd73

sao2								
12090 faults simulated 1024 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	768	0	80	75	0	222	1	3.3667s
2	468	0	24	66	0	95	2	3.3167s
3	744	0	61	73	0	181	1	3.1333s
4	604	0	117	103	0	938	40	5.2167s
5	764	0	65	81	0	209	15	3.3333s
6	752	0	7	64	0	89	1	2.3667s
7	624	0	18	66	0	84	1	2.8167s
8	748	0	68	66	0	110	1	3.1167s
9	689	0	2	64	0	80	1	2.5500s
10	536	0	9	69	0	135	2	3.0000s
Average	669.700	0.000	45.100	72.700	0.000	214.300	6.500	3.222
Std. Dev.	101.151	0.000	36.542	11.367	0.000	246.384	11.902	0.737

Table D.20: Results for Benchmark sao2

bw								
12720 faults simulated 32 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	13	1	216	110	0	3038	400	4.0833s
2	18	0	460	138	0	3665	40	13.0000s
3	18	0	297	127	0	3705	3	6.6167s
4	20	0	334	115	0	2601	3	7.0833s
5	12	0	95	96	0	922	9	1.7833s
6	24	0	540	179	0	6338	356	16.9500s
7	28	0	1651	246	0	14540	155	190.8000s
8	29	0	2522	232	0	22591	663	226.9500s
9	28	1	1941	368	0	49986	38977	180.4667s
10	28	0	1544	261	0	18981	2115	112.9333s
Average	21.800	0.200	960.000	187.200	0.000	12636.700	4272.100	76.067
Std. Dev.	6.145	0.400	823.429	83.187	0.000	14380.389	11584.142	87.145

Table D.21: Results for Benchmark bw

rd84								
25651 faults simulated 256 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	196	0	13	180	1	1131	2	3.3167s
2	114	0	46	188	1	1149	3	4.6833s
3	132	0	120	181	1	1630	31	5.8167s
4	225	0	122	226	1	1512	1	5.8667s
5	180	0	152	207	1	1415	1	6.0833s
6	80	0	22	190	1	1474	1	4.7167s
7	117	0	21	191	1	1159	1	4.0833s
8	79	0	46	211	1	1617	4	5.2167s
9	89	0	9	199	1	1424	1	4.5500s
10	180	0	42	196	1	1150	1	4.1833s
Average	139.200	0.000	59.300	196.900	1.000	1366.100	4.600	4.852
Std. Dev.	49.745	0.000	49.390	13.612	0.000	190.594	8.857	0.845

Table D.22: Results for Benchmark rd84

9sym								
28441 faults simulated 512 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	256	0	4	129	0	815	0	3.9167s
2	163	0	4	126	0	853	2	4.4167s
3	258	0	15	132	0	958	1	4.2333s
4	160	0	4	127	0	787	1	4.6333s
5	182	0	7	129	0	1028	1	4.4500s
6	285	0	15	133	0	863	4	4.2167s
7	359	0	51	147	0	1215	2	4.7667s
8	384	0	91	155	0	1194	5	5.2167s
9	384	0	17	150	0	1074	1	3.7500s
10	383	0	20	140	0	871	2	3.9167s
Average	281.400	0.000	22.800	136.800	0.000	965.800	1.900	4.352
Std. Dev.	87.887	0.000	26.305	9.958	0.000	147.242	1.446	0.423

Table D.23: Results for Benchmark 9sym

c432								
19110 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	749	0	2	103	8	180	1	5.3667s
2	754	0	3	106	8	199	2	5.2833s
3	742	0	2	103	8	180	1	5.3500s
4	417	0	3	103	8	180	2	7.1667s
5	662	0	54	157	8	532	6	7.7667s
6	645	0	3	104	8	180	2	5.7500s
7	644	0	2	103	8	180	1	5.9000s
8	632	0	4	106	8	202	1	5.9500s
9	756	0	4	106	8	208	4	5.4833s
10	727	0	2	106	8	199	1	5.6500s
Average	672.800	0.000	7.900	109.700	8.000	224.000	2.100	5.967
Std. Dev.	97.911	0.000	15.385	15.824	0.000	103.215	1.578	0.792

Table D.24: Results for Benchmark c432

c499								
29403 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	729	0	80	1646	81	960	1	164.3500s
2	751	0	194	1642	81	960	4	159.6667s
3	740	0	124	1641	81	960	2	155.0167s
4	752	0	46	1657	81	965	1	145.3500s
5	756	0	67	1648	81	961	1	153.0000s
6	733	0	57	1649	81	960	1	148.6000s
7	756	0	725	1652	81	2165	2995	253.1833s
8	731	0	582	1637	81	960	25	234.2167s
9	743	0	149	1640	81	960	5	161.5333s
10	748	0	224	1637	81	960	130	172.0000s
Average	743.900	0.000	224.800	1644.900	81.000	1081.100	316.500	174.692
Std. Dev.	9.741	0.000	223.750	6.300	0.000	361.303	893.637	35.520

Table D.25: Results for Benchmark c499

c880								
97903 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	506	0	33	457	75	1607	3	29.8000s
2	655	0	20	452	75	1850	1	26.6500s
3	569	0	70	468	75	1992	3	30.6667s
4	681	0	44	461	75	1773	1	26.2833s
5	757	0	117	469	75	2153	1	28.6833s
6	512	0	18	437	75	824	4	28.3167s
7	720	0	16	459	75	1649	1	25.4333s
8	632	0	104	495	75	2169	2	32.8000s
9	742	0	37	454	75	1671	1	26.0833s
10	724	0	44	466	75	1907	4	25.6667s
Average	649.800	0.000	50.300	461.800	75.000	1759.500	2.100	28.038
Std. Dev.	88.283	0.000	33.785	14.190	0.000	364.030	1.221	2.333

Table D.26: Results for Benchmark c880

c1355								
171991 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	615	0	558	3027	122	117405	1	685.2333s
2	671	0	471	3060	122	104691	1	628.2667s
3	732	0	526	2954	122	112557	6	635.7167s
4	437	0	183	2994	122	115319	1	630.3000s
5	737	0	552	3017	122	126295	3	626.5000s
6	500	0	278	3114	122	131260	37	656.0667s
7	619	0	410	2968	122	108344	5	626.0333s
8	437	0	181	3039	122	123290	1	655.4167s
9	604	0	584	2978	122	111725	5	690.4833s
10	690	0	390	2953	122	124854	1	581.3333s
Average	604.200	0.000	413.300	3010.400	122.000	117574.000	6.100	641.535
Std. Dev.	106.253	0.000	145.329	48.911	0.000	8152.869	10.473	30.226

Table D.27: Results for Benchmark c1355

c1908								
416328 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	741	0	1691	5440	1085	109080	3448	2318.7500s
2	755	0	821	5230	1085	90665	16	1777.2000s
3	745	0	1232	5386	1085	108524	195	2398.5667s
4	740	0	1593	5385	1085	107110	714	2253.4500s
5	748	0	815	5281	1085	103423	252	1802.2667s
6	746	0	1440	5404	1085	110671	108	2512.5333s
7	745	0	816	5263	1085	93843	30	2033.3167s
8	729	0	1511	5457	1085	122454	15717	2247.1167s
9	744	0	1546	5260	1085	108117	30	2526.4333s
10	738	0	308	5200	1085	68520	4	1737.5667s
Average	743.100	0.000	1177.300	5330.600	1085.000	102240.700	2051.400	2160.720
Std. Dev.	6.488	0.000	435.939	88.630	0.000	14025.498	4663.212	287.401

Table D.28: Results for Benchmark c1908

c2670								
1016025 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	543	0	422	2855	954	15784	24	709.0333s
2	742	0	23	2745	954	12280	14	512.5667s
3	746	0	32	2764	954	12298	36	530.0333s
4	727	0	42	2777	954	12271	4	540.0000s
5	571	0	68	2869	954	13389	8	637.9333s
6	751	0	14	2784	954	12282	1	516.8667s
7	722	0	4	2769	954	12282	4	533.1167s
8	736	0	26	2798	954	12278	120	534.5667s
9	742	0	39	2760	954	12294	14	518.6000s
10	558	0	43	2780	954	12342	84	600.2667s
Average	683.800	0.000	71.300	2790.100	954.000	12750.000	30.900	563.298
Std. Dev.	83.417	0.000	118.087	38.571	0.000	1063.172	37.774	61.878

Table D.29: Results for Benchmark c2670

c3540								
1476621 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	540	0	68	3891	2201	85702	8	1292.7833s
2	777	0	247	4038	2201	94138	16	1159.0833s
3	682	0	194	4020	2201	104824	4	1231.0167s
4	580	0	336	3974	2201	87105	8	1339.9333s
5	676	0	81	3891	2201	85708	4	1141.8000s
6	751	0	271	4070	2201	98884	80	1255.7667s
7	778	0	144	3935	2201	84446	2	1104.4333s
8	557	0	129	3934	2201	104619	672	1304.9667s
9	672	0	68	3876	2201	103430	9	1154.9167s
10	767	0	175	3997	2201	87269	12	1152.7667s
Average	678.000	0.000	171.300	3962.600	2201.000	93612.500	81.500	1213.747
Std. Dev.	87.405	0.000	86.808	64.156	0.000	8142.225	198.042	77.362

Table D.30: Results for Benchmark c3540

c5315								
3086370 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	688	0	30	6297	1461	38598	28	2762.5333s
2	735	0	21	6312	1461	38603	12	2630.1333s
3	657	0	24	6315	1461	38666	3	2878.4500s
4	748	0	26	6301	1461	38538	66	2605.4500s
5	733	0	17	6320	1461	38571	15	2860.9833s
6	711	0	32	6301	1461	38547	56	2684.1833s
7	742	0	13	6292	1461	38534	2	2646.8667s
8	614	0	10	6296	1461	38832	12	3101.0000s
9	637	0	17	6305	1461	38768	8	3276.5833s
10	712	0	21	6294	1461	38611	72	2866.5167s
Average	697.700	0.000	21.100	6303.300	1461.000	38626.800	27.400	2831.270
Std. Dev.	44.668	0.000	6.730	9.012	0.000	95.502	25.578	206.694

Table D.31: Results for Benchmark c5315

c7552								
6913621 faults simulated 1000 test vectors								
Pattern	Frequency	F_L	F_P	F_S	F_U	F_{NS}	F_{NP}	CPU Time
1	692	0	92	12566	3199	173231	60	11075.3167s
2	697	0	542	14929	3199	369729	560	14727.0667s
3	654	0	526	12567	3199	185695	5	11352.7833s
4	689	0	307	12599	3199	185020	17	11092.0333s
5	734	0	270	12601	3199	172980	3	10641.7000s
6	693	0	174	12589	3199	186016	20	10289.7833s
7	747	0	347	12617	3199	184664	6	10381.2667s
8	719	0	304	12610	3199	186528	16	10635.1667s
9	551	0	170	12589	3199	180934	54	12071.6000s
10	721	0	401	12578	3199	184752	2	10093.0833s
Average	689.700	0.000	313.300	12824.500	3199.000	200954.900	74.300	11235.980
Std. Dev.	52.599	0.000	140.459	701.684	0.000	56464.823	163.074	1288.111

Table D.32: Results for Benchmark c7552